IDF: A framework for the incremental development and conformance verification of UML active primitive components

Thomas Lambolais [a], Anne-Lise Courbis [a,*], Hong-Viet Luong [b], Christian Percebois [c]

[a] *Laboratoire LGI2P, école des mines d'Alès, Site de Nîmes, Parc Scientifique Georges Besse, 30 035 Nîmes cedex 1, France*
[b] *M2M-NDT, 1 rue de Terre Neuve, Miniparc du Verger, bâtiment H, Les Ulis, 91 940, France*
[c] *IRIT, équipe Macao, université Paul Sabatier, 118 Route de Narbonne, 31 062 Toulouse, cedex 9, France*

Keywords:
State machine refinement
Incremental development
Conformance relations

A B S T R A C T

Modelling component behaviour is widely recognised as a complex task during the specification and design phases of reactive systems. Our proposal for treating this problem involves an incremental approach that allows UML state machines to be built using a composition of two types of development: model extension for adding services or behaviours, and refinement for adding details or eliminating non-determinism. At each step of the development process, the current model is verified for compliance with the model obtained during the previous step, in such a way that initial liveness properties are preserved. The novelty of this work lies in the possibility to combine and sequence both refinement and extension developments. This iterative process is usually not taken into account in conventional refinement relations. This set of development techniques and verification means are assembled into a framework called IDF (Incremental Development Framework), which is supported by a tool, under the acronym IDCM (Incremental Development of Compliant Models), developed herein in addition to the Topcased UML tool.

1. Introduction

Reactive systems are intrinsically difficult to model since they require designers to address the concepts of parallelism, communication, synchronisation, abstraction and non-determinism. Such systems permanently interact with their environment, at speeds determined by this environment (Halbwachs, 1992). Our work concerns the specification and design phases, both of which are intended to define unambiguous behavioural models that characterise the interactions taking place between the system under design and its environment. This paper focuses on model specifications for components, whose behaviour can be modelled by UML state machines. Such components are designed to be integrated into architectures in order to model complex systems. In this context, component model development is emphasised over architectural concerns. It is suggested in this paper that modelling complexity can be best handled through an incremental approach, whereby the behavioural specification of a system is defined step-by-step. We are seeking an incremental design to encompass two development directions: adding details, which is a typical refinement view leading from abstract to more concrete models; and adding new behaviours, services or functionalities, which constitutes an extension mechanism. For these two directions, the development approach is to be accompanied by evaluation techniques so as to verify whether the added details or behaviours actually preserve those previously defined.

We are thus proposing an iterative development process that achieves agile modelling, by combining such a development approach with formal evaluation means. The historical account of agile methods, provided by Larman and Basili (2003), points out that in 1969 B. Randell and F.W. Zurcher argued that: "The basic approach recognizes the futility of separating design, evaluation and documentation processes in software design. The design process is structured by an expanding model [...] It is tested and further expanded through a sequence of models that develop an increasing amount of function and detail. Ultimately, the model becomes the system."

Many works have been proposed by the formal community to offer verification support for system development or UML model development, including the verification of temporal logic properties using model checking techniques and refinement techniques based on theorem proving. Nevertheless, we will show that few evaluation means are suited for what we expect from an incremental development. To the best of our knowledge, no work has yet to formalise and implement the incremental development and verification of behavioural UML models. In previous works (Luong et al., 2008), we have shown

* Corresponding author. Tel.: +33 4 66 38 70 22.
E-mail addresses: thomas.lambolais@mines-ales.fr (T. Lambolais), anne-lise.courbis@mines-ales.fr (A.-L. Courbis), hv.luong@m2m-ndt.com (H.-V. Luong), christian.percebois@irit.fr (C. Percebois).

how to implement a conformance relation between two UML state machines. Then, we have studied and implemented specific relations based on conformance and suitable for incremental development (Luong, 2010; Courbis et al., 2012). New topics addressed in details in this paper are the following:

- an analysis of expected properties of relations used to compare models during their development;
- a survey of conventional refinement relations with respect to these properties;
- the set of relations we have implemented, which constitutes the core of IDF (Incremental Development Framework), along with their complexity analysis;
- the soundness of the transformation of state machines into LTS;
- a presentation of the tool called IDCM (Incremental Development of Compliant Models) we have developed to support IDF.

The present article will be divided into ten sections, including this introduction. Section 2 will introduce the incremental modelling concepts through an example in order to highlight in Section 3 the fundamental properties of IDF relations. These parts will help explain why representative works in this field, as showcased in Section 4, lacks the comprehensiveness to cover all incremental development requirements. Section 5 will then display the formal relations that allow models to be compared with one another, in drawing attention to relations that are relevant to fulfilling incremental modelling requirements. Furthermore, we demonstrate that their composition allows a strategy free incremental development of models. Section 6 will present the implementation of IDF relations. Section 7 will detail the process of transforming UML state machines into LTS prior to their analysis, and Section 8 will expand on the example from Section 2 regarding how incremental relations are applied and will yield results of the model verification step. Section 9 will present IDCM performance results on experimental models. Section 10 will close the article by a discussion of the strengths and weaknesses of IDF and a presentation of our perspectives.

2. Incremental development: introductory example

This section characterises incremental development and illustrates its main concepts by way of example. The core mechanisms for designing complex systems consist of *model refinement* and *model extension*. We are seeking to support a design process by formal behavioural relations that enable the designer to compare early on models obtained at two different steps. According to our process, models are built and verified step-by-step. In focusing on reactive system features, such behavioural comparisons point in particular to the preservation of liveness properties. This is specific to our approach which differs from usual verification works which mainly deal with safety properties.

We observe safety and liveness properties by means of the *interactions* of the system with its environment: accept an *event* (signal or operation reception), or perform an *action* requiring a signal send or operation call. A trace is a partial sequence of interactions starting from the initial state. The following informal definitions are in accordance with the more general definitions and topological characterisation given in Alpern and Schneider (1985, 1987). Although this safety/liveness topology may seem old, this is a complete classification with respect to linear-time (LT) properties: Schneider (1987) demonstrated that any LT property is a conjunction of safety and liveness properties.

Safety properties. A safety property states that some interactions are forbidden for the system after some given traces. This specifies that some traces must not be included in the set of system traces, *e.g.* "the system must not deliver any good if the customer has not paid yet". Safety properties are satisfied by systems whose behaviours are outside of '*must-not behaviours*' in Fig. 1. *P* is a safety property if and only if every violation of *P* occurs after a finite execution of the system (Kupferman and Vardi, 2001). *P* can only be satisfied after infinite executions. Hence, testing approaches can only identify some safety failures.

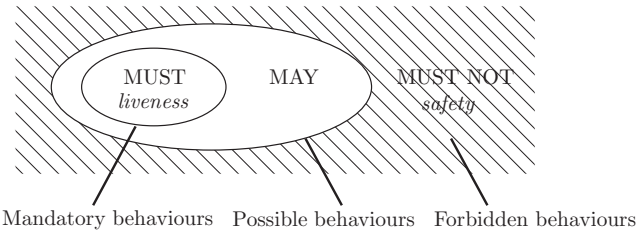


Fig. 1. Liveness and safety properties through 'must', 'may' and 'must not' behaviours.

Liveness properties. A liveness property states that the system will eventually react as it should after some given traces. This specifies that some traces are included in the system trace set, *and* that after these traces, expected actions will eventually be offered, possibly after an unbound delay, *e.g.* "the system will refund the user if he pushes the cancel button". Liveness properties correspond to the *'must behaviour'* set in Fig. 1. We consider that deadlock freedom is a liveness property (Corporation, 2013; Brinksma and Scollo, 1986) since a system is deadlocked when it rejects any input event. *P* is a liveness property if and only if every violation of *P* never occurs after a finite execution. Hence, testing approaches cannot identify liveness failures. Moreover, even if a liveness property *P* is satisfied after a finite execution σ, one should verify that *all* replayed executions σ satisfy property *P*, since we cannot assume that the considered system is deterministic. Testing approaches can neither state liveness property satisfaction. We can nevertheless analyse liveness preservation on models. When reasoning on models, liveness properties can only be established under some fairness assumption.

Fairness assumption. Fairness assumption means that the system is not allowed to continuously favour certain choices at the expense of others (Puhakka and Valmari, 2001). The fairness assumption implies that the system will eventually accept an event occurring infinitely often. Harmless divergences are possible infinite paths of internal actions from which, by fairness hypothesis, the system will eventually exit. On the contrary, critical divergences (or livelocks) are infinite path of internal actions from which the system cannot exit.

2.1. Informal vending machine specification

As an example, let's consider the specification of a vending machine as given below. This specification contains *mandatory parts*, *e.g.* "the machine must refund the customer if the ordered goods are not obtained", as well as *possible parts*, such as options, *e.g.* "goods could be cookies", or services which should be offered most of the time but cannot be guaranteed, *e.g.* "the system delivers drinks, unless it has not been supplied by cups and drinks". Its basic requirements correspond to the most common uses, *e.g.* "the machine is designed to deliver drinks", while secondary requirements would include maintenance functions.

Informal specification of a vending machine

The system delivers goods once the customer has inserted the proper credit into the machine. Goods are mainly drinks but could also be cookies. A technician can shut down the machine with a special code. When used by a customer, the system runs continuously. One important feature is that the system must not cheat the user: if the customer has not inserted enough money, changes his mind or if the system is empty, the machine is to refund the user.

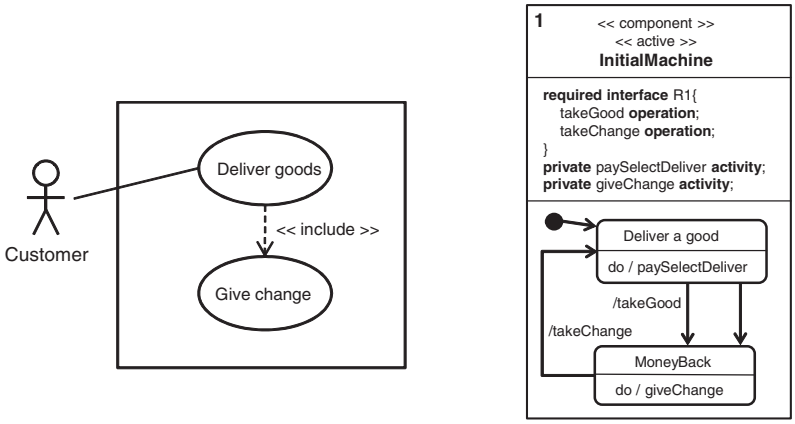


Fig. 2. (a) Use case diagram 1. (b) The InitialMachine active component and associated state machine.

2.2. Initial requirements and models

The incremental development adopts an agile approach. As stated by Scott Ambler (Ambler, 2008), "requirements only need to be good enough: agile software developers do not need a perfect requirement specification, nor a complete one, ... keep refining and completing requirements." At first, only simplified and partial user requirements of the vending machine are considered. These are partial inasmuch as only some external functions appear, *e.g.* maintenance functions do not appear. Moreover, we define *hypotheses* which are requirements on the environment. The system has to hold requirements, provided that *hypotheses* on the environment are satisfied. The incremental approach will progressively remove some hypotheses. Hereafter, we model these requirements by use case diagrams and active components whose behaviour is defined by a state machine. The goal of use case diagrams is to identify the boundaries of the system and to precise the environment related entities. Use case diagrams are useful to see at a glance the designer intents: they highlight new interactions developed at every step which have to be taken into account in the associated state machine.

Hypotheses	
H_1	The technician does not shut down the machine.
H_2	The customer has enough coins.
H_3	There is an unbounded stock of coins to give change.
H_4	The customer does not change his mind.
H_5	Only one generic type of coin is available.
H_6	Good prices are undefined.
H_7	The machine is only used by one customer at a time.

Originating requirements	
R_1	The machine, when supplied, delivers the selected good(s).
R_2	The machine can be used as many times as customers pays.
R_3	The machine does not cheat the customer.

At this point, the machine may be empty if either the cups or coffee is not available. Since the machine cannot cheat the customer, it must provide him with a refund in this case. The use case diagram in Fig. 2a distinguishes customer from machine and identifies two main functions: delivering goods, and returning change. The second function includes refunds if the machine fails to deliver a good. Other actors, such as the technician or an energy source, don't appear. Maintenance operations are not taken into account. Nevertheless, there is no hypothesis stating that the machine is always supplied by goods, so that the service "Deliver goods" may fail.

Use case diagram 1 does not indicate that the machine may be used several times (R_2): all goods may be delivered at once. Fig. 2b

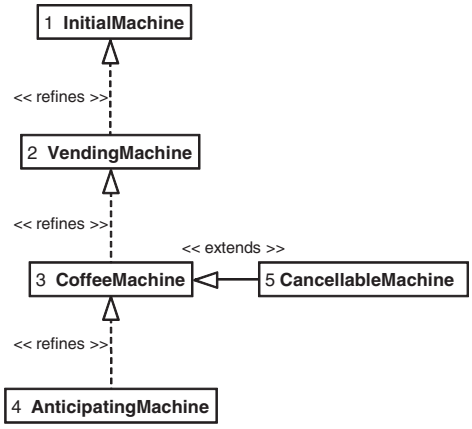


Fig. 3. Outline of the incremental development process of the coffee machine.

presents a simplified active component and its associated state machine, demonstrating that goods are delivered one at a time and moreover that the machine may be used as often as desired. This is a state-oriented description based on activities conducted within these states. The upper compartment of the component compiles declarations relative to operations, signals and activities. At this level of abstraction, the provided interface is not detailed. The procedure required to eventually obtain a good is not detailed at this step. Interactions with the customer only appear through the takeGood and takeChange operations. Both paySelectDeliver and giveChange activities are assumed to lead to termination. There are two transitions from state Deliver a good to state MoneyBack, both of them triggered by the same completion event, one with a takeGood effect and the other without any effect. Hence, this InitialMachine is not deterministic. Non determinism is an abstraction means which enables us, here, to describe a machine which may not be able to deliver a good, without exactly specifying how it fails.

The incremental developments presented herein are diagrammed in Fig. 3; they consist of three refinements and one extension. This diagram reveals the designer's intentions. The "refines" and "extends" relationships are respectively stereotypes of the UML realisation and specialisation relationships. Hence, refinement between state machines is a very specific relation between an abstract state machine and one of its realisations, whereas the extension between state machines corresponds to a precise notion of specialisation and inheritance. At this point, the figure does not indicate

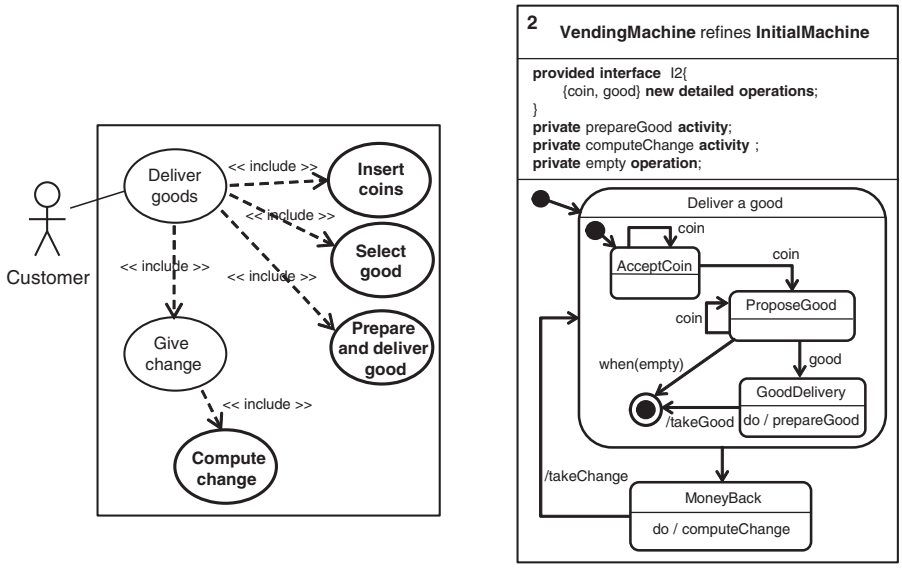


Fig. 4. (a) Use case diagram 2. (b) VendingMachine active component and state machine.

whether the refine and extend relations are actually satisfied or not. Sections 5 and 8 will support the procedure for verifying these relations.

2.3. Refinements

The refinement steps are intended to add details and decrease non-determinism in order to obtain a model closer to the description of a possible implementation model. This development can be visualised along a vertical axis ranging from an abstract to a concrete view.

First refinement. Originating requirements R_1–R_3 from the previous section lack of precision; they are detailed as follows. Each requirement R_i is refined by $R_{i.1}$ to $R_{i.n_i}$, where $R_{i.j}$ may also be refined. The set of hypotheses remains unchanged.

Refined requirements 1	
$R_{1.1}$	The machine accepts coins from the customer.
$R_{1.1.1}$	The machine accepts coins until the product is selected.
$R_{1.1.2}$	The machine can accept an unlimited number of coins.
$R_{1.2}$	The machine offers a selection of goods.
$R_{1.3}$	The machine delivers, when supplied, the selected good.
$R_{2.1}$	The machine initiates a new transaction once the current one is finished.
$R_{3.1}$	The machine returns change to the customer.
$R_{3.1.1}$	If the customer pays more than what is necessary, the machine returns the appropriate change.
$R_{3.1.2}$	If the machine is empty, it refunds the customer in full.

The corresponding refined use case diagram is shown in Fig. 4a; interactions with the customer appear in Fig. 4b. The VendingMachine component presents two notions, which we have led to identifying a special syntax:

(i) the *refinement declaration of a component*. Here, the designer wants VendingMachine to refine InitialMachine. At this step, the *refinement* clause is a declaration of intent, for subsequent verification;

(ii) the *introduction of newly provided and required interfaces*. The operations coin and good of provided interface I2 introduced in Fig. 4b are said to be *new detailed operations*, since they are refinements of unobservable interactions taking place in the InitialMachine machine. The fact that these operations are declared as *new detailed operations* will lead to hiding them when comparing the behaviour of VendingMachine to that of InitialMachine. The coin and good operations represent the means by which a customer controls the VendingMachine: they can be observed at the detailed level of this first refinement, though they are not visible at the first abstract level of details in Fig. 2b.

The change event when(empty) corresponds to an internal, and thus uncontrollable event.

Required properties for incremental development. This first refinement exhibits some of the incremental properties we are seeking to verify: VendingMachine must preserve the liveness properties of InitialMachine. The fairness assumption states that the loop transition with event coin on state AcceptCoin will not be continuously selected which guarantees that state ProposeGood will be reached. The liveness properties of InitialMachine comprise: (i) the machine does not possess any livelock, since the paySelectDeliver and giveChange activities are assumed to terminate; and (ii) the machine does not deadlock, *i.e.* completion events are eventually triggered and the machine can offer goods indefinitely. In VendingMachine, the paySelectDeliver activity is refined by a sub-state machine in the Deliver a good state, which has been designed to terminate.

The prepareGood and computeChange activities and the empty operation are all private. Once again, the absence of a livelock relies upon their termination. We expect VendingMachine to be a correct refinement of InitialMachine. This will be verified in Section 8 using the formal relations proposed in Section 5.

Second refinement. According to this refinement, the machine is specialised as a coffee machine, and two types of coins are proposed. A coffee is assumed to cost 50p. Hypotheses H_5 and H_6 are refined into

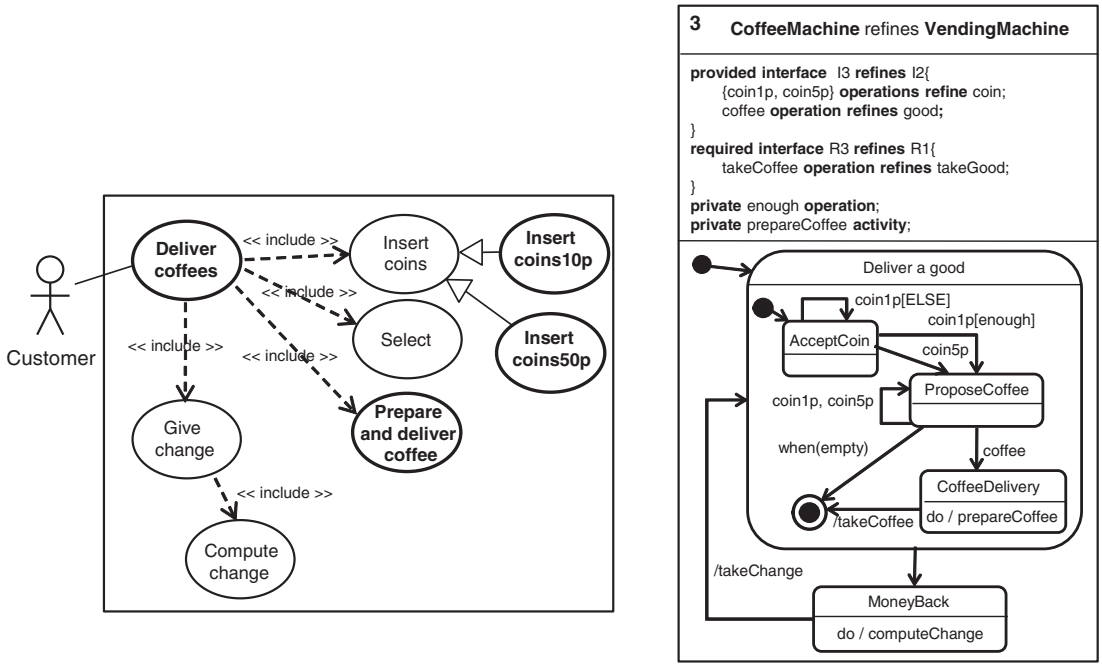


Fig. 5. (a) Use case diagram 3. (b) CoffeeMachine active component and state machine.

$H_{5.1}$ and $H_{6.1}$, while requirements $R_{1.2}$ and $R_{1.3}$ are refined into $R_{1.2.1}$ and $R_{1.3.1}$.

Refined hypotheses	
H_1–H_4	*< unchanged >*
$H_{5.1}$	Coins are either 10p or 50p.
$H_{6.1}$	A coffee costs 50p.
H_7	*< unchanged >*

Refined requirements 2	
$R_{1.1.1}$	The machine accepts 10p and 50p coins.
$R_{1.1.1.1}$	The machine accepts coins until the product is selected.
$R_{1.1.1.2}$	The machine can accept an unlimited number of coins.
$R_{1.2.1}$	The machine only offers coffee.
$R_{1.3.1}$	The machine delivers coffee.

The new use case diagram is presented in Fig. 5a, and b shows the component and state machine.

Required properties for incremental development. In order to compare CoffeeMachine to VendingMachine, the two interfaces must be viewed at the same abstraction level. Hence, the new coin1p and coin5p operations are both replaced by coin, and the coffee operation is replaced by good. All liveness and safety properties of VendingMachine must be preserved.

Third refinement. The requirements remain unchanged, but the designer is seeking to improve the way the machine reacts when empty. The AnticipatingMachine machine in Fig. 6 is designed to run more efficiently.

Required properties for incremental development. Although AnticipatingMachine is declared as a refinement of CoffeeMachine, this refinement should allow detecting a number of errors. First, AnticipatingMachine does not preserve the safety properties of CoffeeMachine since it may initially offer takeChange without any user interaction. This aspect can be observed by considering the AcceptCoin state with the completion event when(empty) followed by the OutOfStock state with the completion event after(10s). Second, AnticipatingMachine does not preserve all the liveness properties since it may initially refuse any coin1p or coin5p actions.

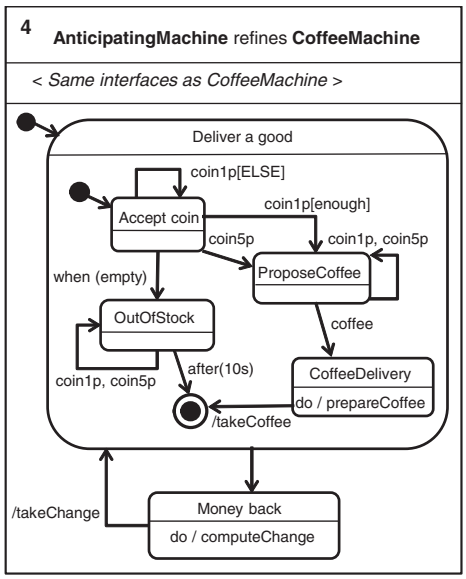


Fig. 6. AnticipatingMachine active component and state machine.

2.4. Extension

Let's now forward hypotheses H_2 and H_4. We want to ensure that the customer is refunded in case he changes his mind or is short of money. Requirement $R_{3.2}$ is newly added.

Extended requirement 1	
$R_{3.2}$	The customer can cancel the transaction and has to be refunded.

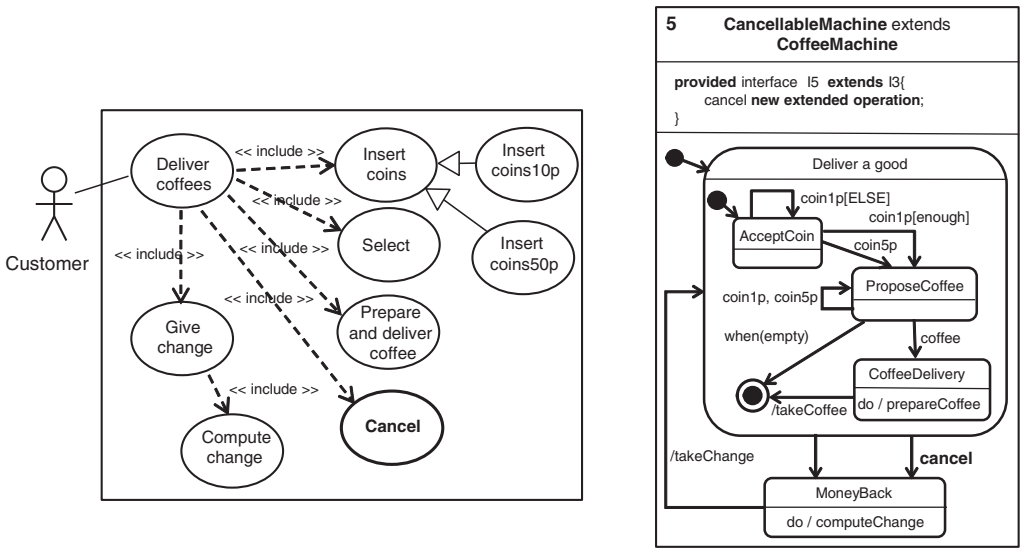


Fig. 7. (a) Use case diagram 4. (b) CancellableMachine active component and state machine.

The corresponding use case diagram, component and state machine are shown in Fig. 7.

Required properties for incremental development. CancellableMachine is designed as a correction of AnticipatingMachine. It is declared as an extension of CoffeeMachine. In the following, traces are denoted using '.' between actions and '*' for unbounded occurrences of actions. CancellableMachine offers new traces, such as cancel.takeChange or coin*.cancel.takeChange, though it must still accept previous behaviours. The liveness properties of CoffeeMachine must be satisfied by CancellableMachine.

2.5. Increment

We also need a third relation when a development directly includes both extension and refinement. In the above example, instead of refining VendingMachine into CoffeeMachine and extending the latter into CancellableMachine, we may have directly incremented VendingMachine into CancellableMachine. This third relation is called "increments". This relation must be larger than extension and refinement relations. Moreover, it must contain any combination of extension and refinement relations in such a way that the final model is expected to be one of a possible system implementation compliant with the initial specification. Hence, we shall need a precise notion of implementation that will be defined in Section 5.

Table 1
Common properties of the extension, refinement and increment relations.

P_1:	*Preorder relations* — They are both reflexive and transitive.
P_2:	*Non-symmetric* — Non-symmetric relations offer more possibilities in iterative developments.
P_3:	*Liveness preservation* — The relations ensure that the new model accepts what the former model was required to accept. This includes new deadlocks and livelocks detection.
P_4:	*Fairness* — Liveness preservation is set up in accordance with the fairness assumption.
P_5:	*Composability* — The composition of extension and refinement relations preserves the properties of the incremental relation.

Table 2
The refinement relation must satisfy properties P_1–P_5 as well as the two properties P_6 and P_7.

P_6:	*Safety preservation* — The relation detects new sequences of operations.
P_7:	*Reduction* — The relation allows removing *optional* behaviours.

3. Incremental development: required properties of IDF relations

An incremental development is a specification and design process that combines both refinements, extensions and increments, as well as a means for their verification. The designer is free to follow several strategies as depicted in Fig. 8. In Fig. 8a, the model is first refined and then extended, whereas in Fig. 8b it is first extended then refined. The most realistic strategy is the mixed one shown in Fig. 8c.

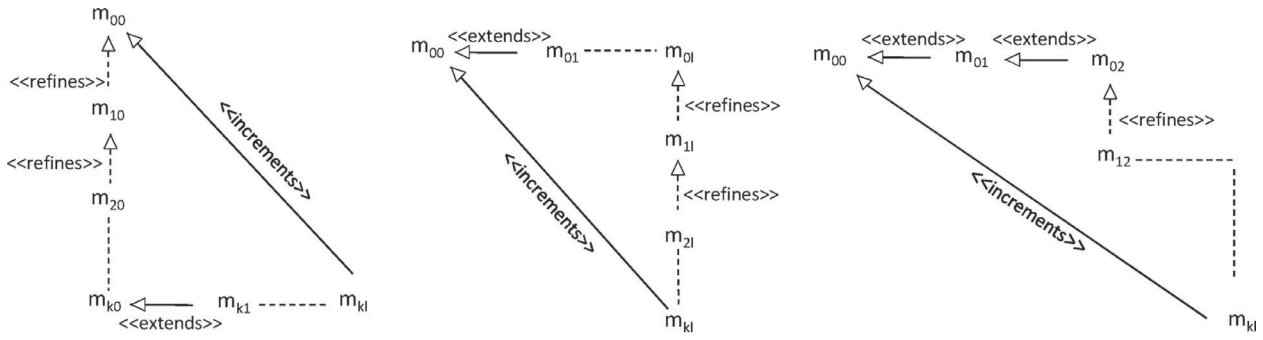
The incremental design offers many advantages. The designer can stop specification development if anticipated basic functions have been defined and then begins system implementation. With this initial version, the designer can generate quick feedback from clients on fundamental functions before finalising the complete model.

The technical feasibility of the solution is addressed prior to model completion; design can thus be controlled in a more reactive manner in order to take into account strategic marketing requirements, or integrate functional requirement modifications suggested by clients.

From the example described above, we gather the properties we have identified for the three relations we are seeking: refinement, extension, and incremental relations. These properties, listed in Tables 1–3, come from theoretical and pragmatical motivations for incremental and iterative modelling approaches of reactive systems. The theoretical background is a mix of formal refinement techniques

Table 3
The extension relation must satisfy properties P_1–P_5 as well as the two properties P_8 and P_9.

P_8:	*Behavioural preservation* — The relation detects the removal of *optional*behaviours.
P_9:	*Extension* — The relation allows for behavioural extension.



(a) Vertical-first strategy. (b) Horizontal-first strategy. (c) Mixed strategy.

Fig. 8. Incremental development strategies.

and theorem proving, such as the B method (Abrial, 1996, 2010), and model checking techniques (Baier and Katoen, 2008; Katoen, 2012). The pragmatical motivations comes from experiences with interactive programming languages, such as the CAML functional language (Bertot and Castéran, 2004; Sagonas, 2013) and also by practical Agile approaches (Ambler, 2008) which enable extensions.

First of all, the three kinds of relations must share common properties, referred by P_1–P_5 in Table 1. Since these relations are used in an iterative development process, they all have to be transitive and reflexive (P_1). We also have to find relations which provide a good balance between a strict verification (strong relations) and a great expressiveness (weak relations). The former will detect[1] too many (irrelevant) errors, the latter will ignore major errors. Symmetric relations are most of the time too strong for this purpose (P_2). For reactive systems, the main kind of properties we focus on are *liveness* properties (P_3): we want the new model to be as reactive as the previous one under *fairness assumption* (3). This enables us to distinguish critical divergences (livelocks) to harmless divergences, from which the system can always eventually exit. Finally, in order to combine refinements and extensions, the required relations must be composable (P_5). We will see that some useful relations satisfy these five properties, which shows that this set of five properties is relevant and consistent.

Within the state machine terminology, 'must behaviours' correspond to liveness properties. In case of refinement relations, we also consider safety properties (P_6), and the relation must be wide enough to remove some non-determinism. This corresponds to ability to remove behaviours which are not mandatory (P_7). Fig. 9 presents a classification of the three sought relations. Note that we have chosen for these verification relations, other notations herein to distinguish them from the modelling relationships introduced in Section 2.

In contrast with classical refinement approaches, we also consider partial models which can be enriched. This corresponds to properties P_8 and P_9. Clearly, P_6 and P_9 properties cannot both be satisfied, as well as P_7 and P_8.

Various studies have previously addressed the problems of model comparison or evaluation-based relations, but only a few of them have actually studied their occurrence in a repetitive development process that guarantees the aforementioned properties. The review of existing works presented in the next section points out this shortcoming and illustrates why often proposed refinement relations are unsuitable for incremental development.

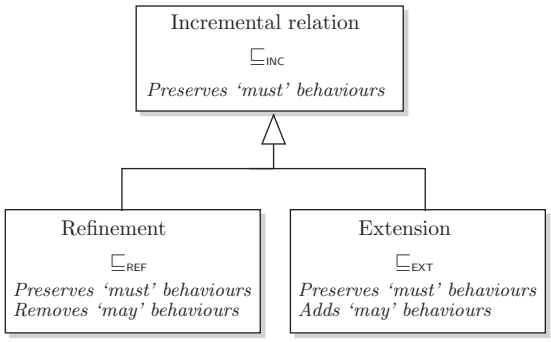


Fig. 9. Classification of researched relations.

4. State-of-the-art of UML state machine construction and verification techniques

There is an increasing number of works dealing with UML model consistency analysis as shown in surveys proposed by Mens et al. (2005) and Khalil and Dingel (2013). These works address a huge set of problems named evolution, co-evolution or refactoring which is larger than our topic of incremental development. We will focus on works dealing with vertical consistencies of behavioural models, that is the consistency of models at different levels of abstraction (Huzar et al., 2005). Most of these works (75% according to Lucas et al. (2009)) set the semantics using a formal language, which has led us to review those works addressing model verification originating from both the UML community and the formal community.

The first sub-section below will highlight the main model checking techniques involved in UML model verification and specifically state machines. Our analysis can be complemented by the review about model checking for UML proposed by Bhaduri (2004) and Usman et al. (2008) and the survey about the formal semantics of UML state machine proposed by Crane and Dingel (2005) and Liu et al. (2013). The second sub-section will indicate why these techniques, even though they are supposed to be refinements, remain inappropriate for the incremental development process. Our argument will be based on the set of properties listed in the previous section.

[1] Given a property X and a binary relation $\mathscr{R}$ over models, the facts that $\mathscr{R}$ detects, preserves, ensures or allows X are defined as follows:

- $\mathscr{R}$ *detects* X if, for all models M and M', when M' satisfies X whereas M does not, $M \mathscr{R} M'$ does not hold.
- $\mathscr{R}$ *preserves* X if, for all models M and M', when M satisfies X, $M \mathscr{R} M'$ entails that M' also satisfies X.
- $\mathscr{R}$ *ensures* X if, for all models M and M', $M \mathscr{R} M'$ implies M' satisfies X.
- $\mathscr{R}$ *allows* X if, for some models M and M', M' satisfies X and $M \mathscr{R} M'$.

4.1. UML state machine analyses

Two main approaches are available: consistency by construction, and *a posteriori* consistency.

Sunyé et al. (2001) and Prochnow et al. (2006) treated consistency by construction. Sunyé et al. (2001) addressed the refactoring problem and proposed a state machine transformation that ensures behaviour preservation. Transformations are expressed through rules modifying the state machine structure. Behaviour preservation is checked by OCL rules, with no demonstration provided about liveness or safety preservation. Moreover, this technique is incompatible with the addition of functionalities and can thus only be applied in a refinement context. Prochnow et al. (2006) defined a number of rules expressed in OCL or Java at the meta-model level, that perform style checking and robustness verification from syntactic and semantic points of view. This approach offers effective support in guiding designers but is not dedicated to model comparison from a behavioural perspective. Pons (2005) proposed an original UML refinement approach qualified as being "formal to informal". She defined refinement patterns expressed on OCL which are inspired from object-Z refinement principles. There is however no significant information about the verification of the refinement.

Most works perform *a posteriori* consistency analyses by comparing the designed model to a reference model. Since UML semantic is ambiguous (Fecher et al., 2005), most of works set this semantic with a formal language (Bhaduri, 2004; Fecher and Schönborn, 2007; Lano, 2009; Schönborn and Kyas, 2010; Liu, S. and Liu, Y. and André, E. and Choppy, C. and Sun, J. and Wadhwa, B. and Dong, J., 2013; Knapp and Mossakowski, 2014). An exception is found in Boiten and Bujorianu (2003) who presented an approach to check the refinement between state machines without applying any translation into a formal language. It is based on an "intuitive" principle of unification. The rationale of the unification principle is highlighted on a simple example by developing it both in UML and Z. This approach has not been defined for general modelling contexts and has not been automated.

The objectives of *a posteriori* consistency approach relies on the ability to utilise trustworthy model checking tools; the consistency analysis modules developed are UML-free and can thus be applied to other modelling languages as well. We have classified works considered herein into three categories, depending on the target formal language. The drawbacks of the techniques involved for incremental purposes will be analysed in the next sub-section. The main techniques are listed below:

- *Set theoretic languages*, which encompass B language (Abrial, 1996), Event-B (Abrial, 2010) and Object-Z (Smith, 2000).
  Truong and Souquieres (2005) performed a transformation from UML behavioural models (state machines and collaboration diagrams) into B language. During refinements, the preservation of system invariants is proved. Said et al. (2009); Said (2010); Said et al. (2013) defined the UML-B language. They propose a translation from UML state machines to Event-B, which allows states to be refined and invariant properties to be checked. Rasch and Wehrheim (2003) translated UML state machines into Object-Z and performed consistency verifications (basically liveness, deadlock and dead-code detection) using the refinement-based model checker FDR (for Failure-Divergence Refinement), which is a refinement relation for the CSP process algebra (Manual FDR2, 2010; Goldsmith and Zakiuddin, 1999). Ruhroth and Wehrheim (2012) proposed co-evolution and Object-Z refinement. Hudon and Hoang (2013) defined the UNIT-B method which aims at developing model via refinement by preserving safety and liveness properties.
- *Process algebras and transition systems*, such as CSP (Hoare, 2004), LOTOS (Bolognesi and Brinksma, 1987), Promela (Holzmann, 1997) and IF (Bozga et al., 2002).
  Chimisliu et al. (2009) translated UML state machines into LOTOS in order to generate test cases using TGV (Jard and Jéron, 2005). LOTOS specifications can be verified with the CADP model checker (Garavel et al., 2011). Many equivalence and preorder relations have been implemented within CADP, including strong, observational, branching and trace equivalences. Lilius and Paltor (1999) and Latella et al. (1999) translated UML state machines into Promela (Holzmann, 1997), so as to use the SPIN model checker. This approach was followed up by Burmester et al. (2004), who devised a way to translate UML state machines into the syntax of the Uppaal model checking tool. Knapp et al. (2002) compared state machines with respect to scenarios expressed in terms of sequence diagrams; properties were verified using the Uppaal model checker. These techniques allow the model to be verified with respect to explicitly defined properties but are unable to compare two versions of state machines. This work used the word *incremental* with a different connotation than ours: their *incremental* verification consists of identifying new components or new interactions and only verifying the discrepancies between the new model and the reference model. Meng et al. (2004) defined refinement using a simulation relation expressed from a co-algebraic view. A very similar definition was given by Kouchnarenko and Lanoix (2006). Schönborn and Kyas (2010) defined refinement patterns on state machines based on a simulation relation, whereby the abstract model simulates the concrete one, with additional conditions ensuring the absence of deadlock. Scholz (2001) defined refinement in terms of design rules based on trace inclusion. Ober et al. (2006) translated state machines into IF expressions and used both the IFx tool and CADP toolbox to check safety properties. Dragomir (2014) proposed a contract-based approach in order to compare SysML state machines with their requirements, taking into account assumptions on the environment which is also described by state machines.
- *Logical languages*, such as temporal logics and description logic languages (Baader et al., 2003).
  First-order logic was introduced in Van Der Straeten et al. (2003) to verify the consistency between sequences of events defined by a state machine and a sequence diagram. Royer (2003) demonstrated an approach for writing and proving temporal logic properties on UML state machines; refinement aspects however have not been taken into account. Gnesi and Mazzanti (2005) presented a state/event-based temporal logic that can be used in a model checking environment. For our purposes, the drawback of such approaches lies in the fact that model versions are not being considered: no support is offered to compare a model *under construction* with a former version. Moreover, such approaches require the designer to develop two models that are conceptually different and therefore require various skills: first a state machine model, then a temporal logic specification of desired behavioural properties. Lowe (2008) considered the relationship between CSP specifications and temporal logic specifications. A refinement-based model checker such as FDR has been introduced to check whether a CSP description satisfies a temporal logic specification.

This review indicates that many of the works using model checkers or theorem provers require designers to *explicitly* express the desired safety and liveness properties in a separate language, which is not our aim. Among the aforementioned techniques found in the first two categories, we have focused on those based on equivalence and preorder relations, which could become candidate techniques for supporting incremental developments. Some properties however are missing. The next sub-section will explain the gaps existing between these relations and the expected incremental relations.

Table 4
Comparison of typical refinement relations according to incremental requirements (property P_5 will be examined in Section 5.5).

		Refinement relations				
		Observational equivalence	Trace inclusion	Ready simulation	Event-B and Stuttering refinement	CSP Failure Divergence Refinement
		$\approx$	$\sqsubseteq_{\mathbf{MAY}}$	$\sqsubseteq_{\mathbf{RS}}$	$\sqsubseteq$	$\sqsubseteq_{\mathbf{FDR}}$
P_1	Preorder	✓	✓	✓	✓	✓
P_2	Non symmetric		✓	✓	✓	✓
P_3	Liveness preservation	✓		✓		✓
	Deadlock detection	✓		✓	✓	✓
	Livelock detection	✓			✓	✓
	Divergence detection	✓		✓	✓	✓
	Critical and harmless divergences distinction	✓		✓		
P_4	Fairness assumption	✓	✓	✓		
P_6	Safety preservation	✓	✓	✓	✓	✓
P_7	Reduction		✓		✓	✓
P_8	Behaviour preservation	✓				
P_9	Extension					

4.2. Refinement relations versus incremental development

The relations we consider for comparing an abstract model with a concrete one, which is the *refined* version, are: Milner's observational equivalence (Milner, 1989), trace inclusion, ready simulation (Bloom et al., 1995), stuttering and Event-B refinements (Kouchnarenko and Lanoix, 2006; Meng et al., 2004; Abrial, 2010), and lastly CSP Failure Divergence Refinement (FDR) (Manual FDR2, 2010).

For refinement purposes, trace inclusion compares whether or not the traces of the concrete model are included in the abstract model traces. Ready simulation is a constrained simulation relation yielding a more *observable* relation than observational equivalence. Bloom et al. (1995) provided a precise definition of the ready simulation. Aldini et al. (2010)presented and compared behavioural equivalences on process algebras. Stuttering refinement is defined on propositional labelled transition systems and on UML state machines; this refinement appears to exhibit the same characteristics as the Event-B refinement (Abrial, 2010). Stuttering refinement is based on a simulation together with additional properties to ensure the absence of deadlock and livelock. Schneider et al. (2014) considered refinement and extension approaches in Event-B. However, as they mentioned, "refinement in Event-B does not require liveness properties to be preserved". Therefore, the CSP semantics they associated to Event-B models does not take failures into account. In a CSP context, the failure preorders introduced in Brookes (1984) differentiate processes by their "failure pairs", which consist of sets of refused actions after observable traces.

Let's now compare these relations with respect to the required properties P_1 through P_9 for our incremental construction. Three aspects of liveness will be detailed: deadlock detection, critical divergence detection, and the ability to distinguish between critical and harmless divergences. Otherwise, it is fair to consider divergence as *harmless*. The composability requirement P_5 is irrelevant here since it concerns the composability of extension and refinement relations that is not addressed in the studied relations; this requirement will be further examined in Section 5.5.

Table 4 summarises the comparisons of these relations. Observational equivalence is one of the strongest relations; it satisfies nearly all our target requirements. This bisimulation however is a symmetric relation and cannot be considered as an incremental development relation. Unfortunately, if we were to only consider the observational *simulation*, then most of its attractive properties would disappear. Observational simulation is similar to trace inclusion, with trace inclusion naturally being the weakest of these relations and rarely considered as a refinement relation. It fails to detect any liveness risk since new deadlocks and any type of divergences may appear. Nevertheless, trace inclusion deserves to be mentioned since it is the simplest and largest relation: in the case of deterministic models, every preorder relation considered coincides with the trace inclusion. The other three relations are all refinements. Ready simulation offers an improvement over observational simulation, when combining a condition that makes it coincide with a simulation plus a refusal predicate; it detects some liveness properties, but not livelocks. Stuttering and Event-B refinements improve the ready-simulation by detecting livelocks, though they are not in accordance with fairness hypotheses and cannot distinguish between critical and harmless divergences. Lastly, the CSP refinement detects all liveness violations but considers divergences as dangerous.

All these relations can be considered as *classical refinements*; their goal is to reduce the level of non-determinism and to introduce refining details. It comes as no surprise therefore that none of these relations can be used in an incremental approach for the extension requirement. For refinement purposes, the ready simulation is close to being the refinement relation we are seeking, except for the fact that it is too strong: it does not allow for optional behaviours to be removed. From this study, we can also state that none of the other three preorder relations ($\sqsubseteq_{\mathbf{MAY}}$, $\sqsubseteq$ and $\sqsubseteq_{\mathbf{FDR}}$) are proper refinement relations for the liveness preservation and fairness requirements.

Next section presents relations we have selected to fulfil requirements P_1 to P_9.

5. Formal definitions of IDF relations

Since the relations we are seeking must all preserve liveness under fairness assumption, we have studied a specific relation, whose lonely goal is to preserve liveness. This relation is conformance relation conf (Cleaveland and Steffen, 1990; Leduc, 1992). Although it is not transitive, it is a convenient means to define the three required

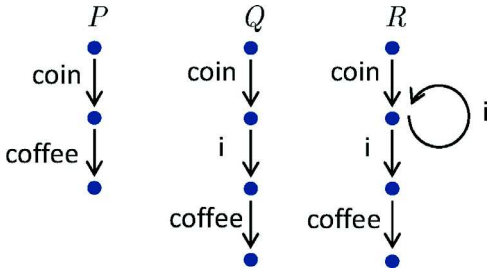


Fig. 10. P, Q and R are not distinguishable with the conf relation.

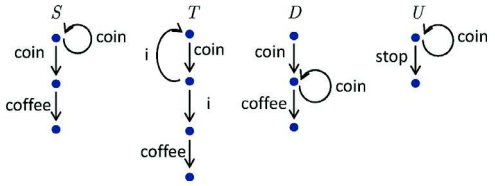


Fig. 11. Benefits of the conf relation over observational equivalence, observational preorder and trace equivalence.

relations for IDF. This section aims at explaining and formally defining these relations on labelled transition systems (LTS).

5.1. Labelled Transition Systems and acceptance sets

We have adopted the following notations for the typical LTS notions (Milner, 1989; 1999) and definitions related to conformance (Cleaveland and Steffen, 1990; Leduc, 1992; Tretmans, 1999).

Let $Act = \mathcal{L} \cup \{\tau\}$ be the set of all actions, where $\mathcal{L}$ is the set of observable actions, and τ the internal action. Moreover, let $\mathscr{P}$ be the set of all names of states or processes.

Definition 1 (Labelled Transitions Systems)**.** (Milner, 1999) An LTS $\langle \mathcal{P}, A, \rightarrow, P \rangle$ is a tuple of a non empty set $\mathcal{P} \subseteq \mathscr{P}$ of states, a set $A \subseteq Act$ of names of actions where $A = L \cup \{\tau\}$, with $L \subseteq \mathcal{L}$ the set of visible actions of the LTS, a relation of transitions $\rightarrow \subseteq \mathcal{P} \times A \times \mathcal{P}$, and an initial state $P \in \mathcal{P}$.

The LTS $\langle \mathcal{P}, A, \rightarrow, P \rangle$ is also designated by P. Let P and P' be LTS, a, $a_{i,\,0 \leq i \leq n}$ actions of Act and $\sigma \in \mathcal{L}^*$ a sequence of observable actions. Next, let's define:

$$
\begin{aligned}
P \xrightarrow{a} P' \quad & =_{def} (P, a, P') \in \rightarrow \\
P \stackrel{\varepsilon}{\Longrightarrow} P' \quad & =_{def} P = P' \text{ or } P \xrightarrow{\tau \cdots \tau} P' \\
P \stackrel{a}{\Longrightarrow} P' \quad & =_{def} \exists P_1, P_2.\ P \stackrel{\varepsilon}{\Longrightarrow} P_1 \xrightarrow{a} P_2 \stackrel{\varepsilon}{\Longrightarrow} P' \\
P \stackrel{a_1 \cdots a_n}{\Longrightarrow} P' \quad & =_{def} \exists P_0, \ldots, P_n.\ P = P_0 \stackrel{a_1}{\Longrightarrow} \cdots \stackrel{a_n}{\Longrightarrow} P_n = P' \\
P \stackrel{\sigma}{\Longrightarrow} \quad & =_{def} \exists P'.\ p \stackrel{\sigma}{\Longrightarrow} P' \\
\text{Traces} : Tr(P) \quad & =_{def} \{\sigma \in L^* \mid P \stackrel{\sigma}{\Longrightarrow}\} \\
P \text{ after } \sigma \quad & =_{def} \{P' \mid P \stackrel{\sigma}{\Longrightarrow} P'\} \\
Out(P) \quad & =_{def} \{a \in L \mid P \xrightarrow{a}\}
\end{aligned}
$$

The conformance relation will be defined with acceptance sets, as proposed by Hennessy (1988). Acceptance sets capture both what a process may accept and what it must accept; they offer a convenient means for observing if one process is more deterministic than another.

Definition 2 (Acceptance set)**.** $Acc(P, \sigma) =_{def} \left\{X \mid \exists P' \in P \text{ after } \sigma,\ \text{such that } X = Out(P')\right\}$

For instance, LTS S and T of Fig. 11 have both the same acceptance set after coin: $Acc(S, \text{coin}) = Acc(T, \text{coin}) = \{\{\text{coin}\}, \{\text{coffee}\}\}$. This means that after coin, they must accept one of these two actions, coin or coffee, but not both. This acceptance set also indicates that they must refuse any other action, which is a safety property. Acceptance sets are compared by the following preorder $\subset\subset$, where $A \subset\subset B$ means that A is less non-deterministic than B.

Definition 3 (Non-determinism preorder)**.** Let A and $B \subseteq \mathcal{P}(L)$, $A \subset\subset B$ if, for all a in A, there exists b in B such that $b \subseteq a$.

For example, for the two LTS S and D of Fig. 11, $Acc(S, \text{coin}) = \{\{\text{coin}\}, \{\text{coffee}\}\}$ and $Acc(D, \text{coin}) = \{\{\text{coin}, \text{coffee}\}\}$; $Acc(D, \text{coin}) \subset\subset Acc(S, \text{coin})$ points out that D is more deterministic than S.

5.2. The conformance relation: the reference implementation relation

Conformance testing methodologies proposed by ISO and ETSI (ISO/IEC9646, 1991) are designed to compare an implementation model with a standard specification. Standard specifications or recommendations serve to define both the mandatory and optional parts. The main idea behind conformance is to verify agreement between an implementation and its specification on required parts; informally speaking, an implementation conforms to a standard if it has properly implemented all *mandatory parts* of the standard (Moseley et al., 2006). In our framework, conformance will be taken as the reference relation to assess if a model implements a specification in the right way.

Conformance leads to the verification that the implementation model '*must feature*' all behaviours required by the specification. A precise definition will be needed not only of what it means to exhibit a behaviour (accept and/or offer, from an observational point of view) but also of how the *may* and *must* parts can be described through non-determinism.

The reference implementation relation between LTS, $\text{conf} \subseteq \mathcal{P} \times \mathcal{P}$, can now be defined by means of acceptance sets, as follows:

Definition 4 (Conformance relation conf)**.** Q conforms P, written Q conf P, if for all $\sigma \in Tr(P).\ Acc(Q, \sigma) \subset\subset Acc(P, \sigma)$.

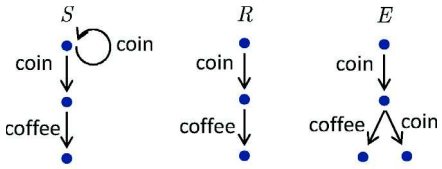
The conf relation is comparable, in a mathematical sense, to Milner's bisimulations (Milner, 1989), as stated by the following proposition:

Proposition 5.1. *Milner's equivalence relations (strong equivalence* $\sim$*, observational congruence* $=$ *and observational equivalence* $\approx$*) are in fact conformance relations:* $\sim \subseteq = \subseteq \approx \subseteq$ conf.

Hence, in observational terms, equivalent processes are also conforming processes. The conformance relation considers fairness in the same way as observational bisimulation. In particular, P, Q and R in Fig. 10 are such that $P = Q = R$, hence the conformance relation is applicable in both directions for these three processes. After coin, these three processes will eventually accept the coffee action as well.

The fact that P conf R shows that some livelocks are harmless. This finding distinguishes conf (and relations stronger than conf) not only from the CSP refinement relations (Bolton and Davies, 2002; Hoare, 2004), in which any livelock is considered dangerous, but also from failure preorders (Leduc, 1995).

The benefit of conf over observational equivalence is its lower discrimination than Milner's observational congruence ($=$). In Fig. 11 for instance, we derive S conf T, T conf S, but $S \neq T$. From an observational standpoint, nothing distinguishes S from T. The primary benefit of the conf relation lies in detecting non-determinism for the purpose of identifying whether or not liveness properties are being preserved. Once again, in the example shown in Fig. 11, process S may refuse coffee after coin^+, a non-empty unbounded occurrences of coins, whereas process D, which is deterministic, cannot. S and D are trace equivalent, yet not in conformance. Finally, the conformance relation supports trace extension and/or reduction: U conf S, whereas U never offers a coffee action but instead offers a new stop action.



R conf S and E conf R, but E c$\not$onf S

Fig. 12. Example of a reduction followed by an extension.

5.3. Extension and reduction relations

Extension and reduction relations are defined as extending or reducing traces, while preserving conformance. They are defined in Brinksma and Scollo (1986) and denoted ext and red. $P \sqsubseteq_{\mathsf{RED}} Q$ (resp. $P \sqsubseteq_{\mathsf{EXT}} Q$) means that P is reduced by Q, or Q red P (resp. P is extended by Q, or Q ext P). These relations are defined as follows:

Definition 5 (Reduction relation $\sqsubseteq_{\mathsf{RED}}$)**.** P is reduced by Q, written $P \sqsubseteq_{\mathsf{RED}} Q$, if $Tr(Q) \subseteq Tr(P)$ and Q conf P.

Definition 6 (Extension relation $\sqsubseteq_{\mathsf{EXT}}$)**.** P is extended by Q, written $P \sqsubseteq_{\mathsf{EXT}} Q$, if $Tr(P) \subseteq Tr(Q)$ and Q conf P.

In Fig. 12, process R is a reduction of S, hence R conf S is also derived. Process E is an extension of R, although it does not conform to S, since E refuses coin and coffee after the trace coin.coin, whereas S must accept either coin or coffee.

5.4. Incremental relations

Fig. 12 reveals that process R cannot be used if the underlying goal is to pursue the development: we may extend R and generate a process E, but E is not a correct implementation of S. The goal of an incremental relation is for any implementation of an incremented process to also be an implementation of the initial process. Consequently, we are specifically seeking a relation that satisfies the following property:

Definition 7 (Incremental relation $\sqsubseteq_{\mathsf{INC}}$)**.** R is incremented by S, written $R \sqsubseteq_{\mathsf{INC}} S$, if for all I, I conf $S \Rightarrow I$ conf R.

The $\sqsubseteq_{\mathsf{INC}}$ relation has been defined by Leduc (1991) and is denoted confrestr, where $R \sqsubseteq_{\mathsf{INC}} S = S$ confrestr R. This relation is our reference *incremental relation*. The $\sqsubseteq_{\mathsf{INC}}$ relation enjoys the two following properties:

- $\sqsubseteq_{\mathsf{INC}} \subseteq \mathsf{conf}^{-1}$. This means that $\sqsubseteq_{\mathsf{INC}}$ can be used as an implementation relation. Since it is transitive, any increment of an incremented model will also lead to a valid implementation. This assessment enables us to pursue a development process at any time.
- If $P \sqsubseteq_{\mathsf{INC}} Q$, for any trace σ of P that is not a trace of Q, then P must refuse everything after σ.

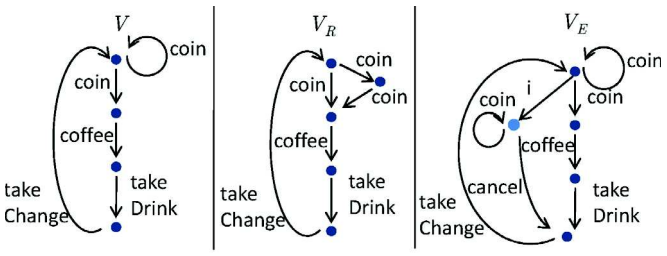
The extension relation is an incremental relation, *i.e.* $\sqsubseteq_{\mathsf{EXT}} \subseteq \sqsubseteq_{\mathsf{INC}}$. As shown in Fig. 12, the reduction relation is not incremental. We have thus defined a new relation $\sqsubseteq_{\mathsf{REF}}$ as the part of $\sqsubseteq_{\mathsf{RED}}$ that is an incremental relation.

Definition 8 (Refinement relation $\sqsubseteq_{\mathsf{REF}}$)**.** $\sqsubseteq_{\mathsf{REF}} =_{def} \sqsubseteq_{\mathsf{RED}} \cap \sqsubseteq_{\mathsf{INC}}$.

The following property is sufficient to characterise $\sqsubseteq_{\mathsf{REF}}$:

- $P \sqsubseteq_{\mathsf{REF}} Q$ if $P \sqsubseteq_{\mathsf{RED}} Q$ and for any trace σ of P that is not a trace of Q, then P must refuse everything after σ.

Fig. 13 displays two examples of incremental developments for a simple vending machine V. The machine V_R is a refinement of V, which in turn limits the possible pricing to just one or two coins.



$V \sqsubseteq_{\mathsf{REF}} V_R \quad V \sqsubseteq_{\mathsf{EXT}} V_E \quad V \sqsubseteq_{\mathsf{INC}} V_R \; V \sqsubseteq_{\mathsf{INC}} V_E$

Fig. 13. A process V with one refinement and extension.

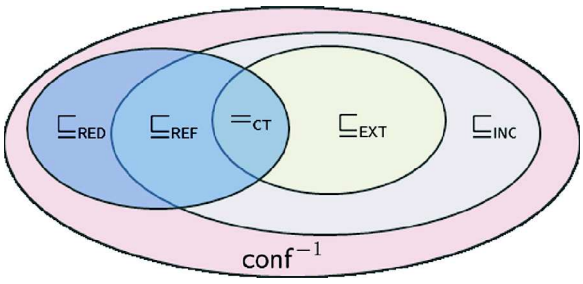


Fig. 14. Position of incremental relations included in $\sqsubseteq_{\mathsf{INC}}$ with respect to the conformance relations included in conf^{-1} (Leduc, 1992).

This reduced machine may accept fewer actions, yet must still accept whatever V is required to accept. The extended machine V_E adds the possibility of being empty, therefore refusing the coffee button forever in this case but still accepting coin action, which is the only mandatory action after the coin* trace.

Lastly, let's define $=_{\mathsf{ct}}$ as the refinement equivalence relation.

Definition 9 (Refinement equivalence relation $=_{\mathsf{ct}}$)**.** $=_{\mathsf{ct}} =_{def} \sqsubseteq_{\mathsf{RED}} \cap \sqsubseteq_{\mathsf{EXT}}$.

We also have $=_{\mathsf{ct}} = \mathsf{conf} \cap =_{Tr}$. The $=_{\mathsf{ct}}$ relation can be qualified as a strict relation for both realisation and abstraction in that it allows the incremented model to be more abstract or more concrete without deleting optional functions or adding new functions. Fig. 14 provides a synthesis of the set of conformance and IDF relations in addition to their inclusion. Table 5 summarises their properties.

Property P_5, which focuses on the composition between incremental relations, is presented in the following sub-section. All the preorder relations included in conf display the following results, which enable reducing the complexity of the conformance relation computation.

Theorem 1. *For any relation $\sqsubseteq$ among $\{=_{\mathsf{ct}}, \sqsubseteq_{\mathsf{REF}}, \sqsubseteq_{\mathsf{EXT}}, \sqsubseteq_{\mathsf{INC}}\}$, any process P, Q, P' and Q', if $P' \approx P$ and $Q' \approx Q$, then*

$$P \sqsubseteq Q \Longleftrightarrow P' \sqsubseteq Q'.$$

Corollary 1. *For any relation $\sqsubseteq$ among $\{=_{\mathsf{ct}}, \sqsubseteq_{\mathsf{REF}}, \sqsubseteq_{\mathsf{EXT}}, \sqsubseteq_{\mathsf{INC}}\}$ and any process P and Q:*

$$P \sqsubseteq Q \Longleftrightarrow \mathrm{minobs}(P) \sqsubseteq \mathrm{minobs}(Q)$$

where minobs(P) is the smallest process observationally equivalent to P.

These results are a direct consequence of both Proposition 5.1 and the transitivity of preorder relations. Indeed, $P' \approx P \Rightarrow P' \sqsubseteq P$ and $Q' \approx Q \Rightarrow Q \sqsubseteq Q'$. From $P' \sqsubseteq P \sqsubseteq Q \sqsubseteq Q'$, we conclude that $P' \sqsubseteq Q'$.

5.5. Composability of IDF relations

We have defined a set of relations to develop models, according to refinement or extension, aimed at applying several development

Table 5
Analysis of conformance relations for incremental developments.

		Conformance	Reduction	Refinement equivalence	IDF relations		
					Refinement	Extension	Incremental relation
		$S' \text{ conf } S$	$S \sqsubseteq_{\mathsf{RED}} S'$	$S =_{\mathsf{ct}} S'$	$S \sqsubseteq_{\mathsf{REF}} S'$	$S \sqsubseteq_{\mathsf{EXT}} S'$	$S \sqsubseteq_{\mathsf{INC}} S'$
P_1	Preorder		✓	✓	✓	✓	✓
P_2	Non symmetric	✓	✓		✓	✓	✓
P_3	Liveness preservation	✓	✓	✓	✓	✓	✓
P_4	Fairness assumption	✓	✓	✓	✓	✓	✓
P_5	Composability			✓	✓	✓	✓
P_6	Safety preservation		✓	✓	✓		
P_7	Reduction	✓	✓		✓		✓
P_8	Behaviour preservation			✓		✓	
P_9	Extension	✓				✓	✓

Table 6
Results of IDF refinement relation compositions.

$\circ$	$\sqsubseteq_{\mathsf{EXT}}$	$=_{\mathsf{ct}}$	$\sqsubseteq_{\mathsf{INC}}$
$\sqsubseteq_{\mathsf{REF}}$	$\sqsubseteq_{\mathsf{INC}}$	$\sqsubseteq_{\mathsf{REF}}$	$\sqsubseteq_{\mathsf{INC}}$
$=_{\mathsf{ct}}$	$\sqsubseteq_{\mathsf{EXT}}$	$=_{\mathsf{ct}}$	$\sqsubseteq_{\mathsf{INC}}$
$\sqsubseteq_{\mathsf{INC}}$	$\sqsubseteq_{\mathsf{INC}}$	$\sqsubseteq_{\mathsf{INC}}$	$\sqsubseteq_{\mathsf{INC}}$

strategies, as highlighted in Fig. 8. We must therefore verify the composition of these relations in order to guarantee that the relation remains incremental. It corresponds to property P_5 which guarantees that the development of models can be iterative and strategy-free.

We need to check that $\mathsf{inc}_1 \circ \mathsf{inc}_2 \subseteq \sqsubseteq_{\mathsf{INC}}$ where inc_1 and inc_2 represent any incremental relations $\sqsubseteq_{\mathsf{INC}}$, $\sqsubseteq_{\mathsf{REF}}$, $\sqsubseteq_{\mathsf{EXT}}$ or $=_{\mathsf{ct}}$. Let's start by studying the composition of any preorder relations.

Proposition 5.2. *Let A be a preorder and X a relation such that $X \subseteq A$, we then have $A \circ X = X \circ A = A$.*

Since $=_{\mathsf{ct}} \subseteq \sqsubseteq_{\mathsf{EXT}}$, we conclude that $\sqsubseteq_{\mathsf{EXT}} \circ =_{\mathsf{ct}} = =_{\mathsf{ct}} \circ \sqsubseteq_{\mathsf{EXT}} = \sqsubseteq_{\mathsf{EXT}}$. This finding suggests that relations $\sqsubseteq_{\mathsf{EXT}}$ and $=_{\mathsf{ct}}$ involved in local refinements guarantee a result that implements the specification with relation $\sqsubseteq_{\mathsf{EXT}}$. The same reasoning can be applied to $\sqsubseteq_{\mathsf{REF}}$ and $=_{\mathsf{ct}}$: $\sqsubseteq_{\mathsf{REF}} \circ =_{\mathsf{ct}} = =_{\mathsf{ct}} \circ \sqsubseteq_{\mathsf{REF}} = \sqsubseteq_{\mathsf{REF}}$. In similar manner, since $\sqsubseteq_{\mathsf{EXT}} \subseteq \sqsubseteq_{\mathsf{INC}}$, $=_{\mathsf{ct}} \subseteq \sqsubseteq_{\mathsf{INC}}$ and $\sqsubseteq_{\mathsf{REF}} \subseteq \sqsubseteq_{\mathsf{INC}}$, the composition of these relations with $\sqsubseteq_{\mathsf{INC}}$ is included in $\sqsubseteq_{\mathsf{INC}}$.

Proposition 5.3. $\sqsubseteq_{\mathsf{REF}} \circ \sqsubseteq_{\mathsf{EXT}} = \sqsubseteq_{\mathsf{EXT}} \circ \sqsubseteq_{\mathsf{REF}} = \sqsubseteq_{\mathsf{INC}}$.

The proofs of these two propositions are straightforward and can be found in (Luong, 2010). This set-up implies that relations $\sqsubseteq_{\mathsf{EXT}}$ and $\sqsubseteq_{\mathsf{REF}}$ involved in local refinements guarantee a result that implements the specification with relation $\sqsubseteq_{\mathsf{INC}}$.

Table 6 provides a summary of the combinations of local refinements and extensions. Since the resulting relation of any composition is included in $\sqsubseteq_{\mathsf{INC}}$, it can be concluded that any sequence of refinements and extensions leads to an incremental relation with no restrictions being placed on incremental relation sequences. This outcome proves that property P_5 is preserved.

6. Implementation and complexity of IDF relations

Even though the conformance and refinement relations have been defined by Brinksma and Scollo (1986) and Leduc (1992), we are still not aware of any published method to compute them. We have thus proposed two algorithms based on the reduction relation to implement these two relations. We will provide an initial overview of the method for computing the reduction relation and then a second overview of the method for computing the extension relation based on the conformance algorithm.

6.1. Implementation of relations

Implementation of reduction and extension preorders. The relation $\sqsubseteq_{\mathsf{RED}}$ is useful for computing conf and $\sqsubseteq_{\mathsf{REF}}$ as will be shown hereafter; furthermore, it is based on the same principle as $\sqsubseteq_{\mathsf{EXT}}$, which is an incremental relation. In order to compute the reduction relation, we studied a very similar concept: the *must* preorder, defined in Cleaveland and Hennessy (1993). Consequently, reduction can be computed as a simulation between acceptance graphs, in addition to a verification of acceptance set inclusion. An acceptance graph is defined as a deterministic LTS, wherein the states are associated with acceptance sets. Similarly, an extension is the opposite simulation between acceptance graphs, while being combined with the same inclusion of acceptance sets.

Theorem 2. *Let P and Q be two LTS whose acceptance graphs are respectively T and U:*

1. $P \sqsubseteq_{\mathsf{RED}} Q \Longleftrightarrow T$ strongly simulates U and $Acc(u) \subset\subset Acc(t)$, *for every pair (t, u) of simulated states.*
2. $P \sqsubseteq_{\mathsf{EXT}} Q \Longleftrightarrow U$ strongly simulates T and $Acc(u) \subset\subset Acc(t)$, *for every pair (t, u) of simulated states.*

Let's now refer to our work (Luong, 2010) on precise definitions of acceptance graphs and the simulation between them, as well as the corresponding propositions that allow implementing reduction and extension relations. For a better understanding, the concepts of acceptance graph and inclusion of acceptance sets are illustrated in Section 8.

These developments are suited to the preorder relations (reduction and extension), which are stronger than the conformance relation, as shown in Fig. 14, though not for the conformance relation itself. The conformance relation is implemented as follows and serves as a basis for the incremental relation.

Implementation of the conformance and refinement relations. It is known (Leduc, 1992) that $\mathsf{conf}^{-1} = \sqsubseteq_{\mathsf{EXT}} \circ \sqsubseteq_{\mathsf{RED}}$, which means that $Q \text{ conf } P \Leftrightarrow \exists R \textit{ such that } P \sqsubseteq_{\mathsf{EXT}} R \wedge R \sqsubseteq_{\mathsf{RED}} Q$. The goal then consists of finding an R process of this type. In Luong et al. (2008) and Luong (2010), we have shown that the smallest extension of P and Q, written $Merge(P, Q)$, is always adequate, *i.e.*:

Theorem 3. *Let P and Q be two LTS. $Q \text{ conf } P \Longleftrightarrow Merge(P, Q) \sqsubseteq_{\mathsf{RED}} Q$.*

The reader can refer to Khendek and von Bochmann (1995) for a precise definition of the *Merge* operator. The demonstration of this theorem is given in Luong et al. (2008). Hence, the conformance relation is verified by a simulation of an acceptance graph and a merge acceptance graph combined with trace inclusion. For the refinement relation $\sqsubseteq_{\mathsf{REF}}$, the implementation step is performed using the property presented above, *i.e.* $P \sqsubseteq_{\mathsf{REF}} Q \Leftrightarrow P \sqsubseteq_{\mathsf{RED}} P \wedge \forall \sigma \in$

$Tr(P) - Tr(Q), Acc(P, \sigma) = \{\varnothing\}$, which indicates that the refinement algorithm is the same as the reduction algorithm, in adding the following verification: any action accepted by the reference model that has not been defined in the refined models leads to a stop.

Implementation of the incremental relation. For the incremental relation $\sqsubseteq_{\mathsf{INC}}$, the implementation step is performed using the property presented above, *i.e.* $P \sqsubseteq_{\mathsf{INC}} P \Leftrightarrow Q \text{ conf } P \wedge \forall \sigma \in Tr(P) - Tr(Q), Acc(P, \sigma) = \{\varnothing\}$.

6.2. Complexity of incremental development relations

Relations for incremental development are based on a strong simulation between two acceptance graphs and the verification of acceptance set inclusion. Let us consider two LTS to be compared having at most n states and m transitions. The complexity of a strong simulation is quadratic in n and m (Fernandez and Mounier, 1992), that of inclusion of acceptance sets is polynomial. Building acceptance graphs is a problem of conversion of a non-deterministic finite automaton to a deterministic finite automaton (Cleaveland and Hennessy, 1993). It is a PSPACE-complete problem, whose time complexity in worse case is exponential (Van Glabbeek and Ploeger, 2008). Cleaveland and Hennessy (1993) pointed out that, in practice, acceptance graphs have less states than the LTS they are generated from. There are some solutions to deal with this complexity. Let us first note that in the case of LTS generated from state machines, we only have small LTS, *i.e.* of the same size order as state machine sizes. Secondly, we can take advantage of computing at first a minimisation with respect to the observational equivalence. The complexity of observational minimisation is linear. If both minimised LTS are observationally equivalent, any conformance preorder is satisfied. Otherwise, the conformance preorders can be computed on minimised LTS as proposed in Corollary 1 in Section 5.4.

7. Transformation of state machines into LTS

The following sub-sections present the UML concepts taken into account and the LTS semantics we have associated to state machines. We will discuss the complexity and correctness of the transformation. Data and time are not represented in LTS, so that we cannot expect properties related to data and time to be analysed on LTS. Nevertheless, the proposed transformation has to guarantee that any liveness and safety property (unvalued and untimed) of a LTS is a property of its originating state machine. This will be checked in the last Section 7.3.

7.1. UML component and state machine features

We will focus on synchronous communication to ease the reading of the article. Asynchronous communications such as signal transmissions and receptions are represented by synchronous mechanism using buffers modelled by LTS. By this way, there is no restriction to automate the transformation of asynchronous communications. In referring to the UML standard, the state machines involved herein are of the *behavioural state* type, without a *history state*. Composite states having concurrent regions are restricted to independent regions without internal synchronisation. They are modelled by parallel LTS which are synchronised on the provided and required operations. The transformation of concurrent regions is not presented in detail to ease the reading of the article. States can be of either the *simple* or *submachine state* type, *i.e.* associated with a state machine.

From the entire range of *pseudostates*, let's consider just the *initial pseudostates*, *entry points* and *exit points*. The transitions may contain a *trigger* of either the *call event*, *time* or *change event* type.

Guards are transformed from an abstract point of view, without taking any data constraints into account. In other words, their expression is free. In order to indicate whether all possible switches have been taken into account, one guard however can be labelled with the keyword ELSE.

An effect may be associated with transitions. We have opted to model effects by *activity diagrams*, whose nodes are presently limited to *call operations* and *control nodes* of the *initial*, *activity final* and *flow final* types. The activity diagram may represent two configurations: a sequence of actions, or an infinite loop of actions.

7.2. Transformation rules

Let *SM* denote the set of state machines. The rules presented here define a relation $t \subseteq SM \times \mathscr{P}$ between state machines and LTS. The rules and their compositions are defined in such a way, that we shall prove relation t to be a function. The transformation goal is twofold: reveal communication sequences between the component and its environment, and hide actions that are irrelevant to the communication. Transformation, based on the set of patterns defined in Fig. 15, is performed automatically.

Hiding consists of representing actions by internal actions labelled τ, written 'i' in the CADP LTS textual format. Below is the list of abstracted state machine concepts:

- guards (cases 4 and 5 in Fig. 15): data used to base the choice are not depicted; the LTS indicates that several paths can be traced from the state representing the choice;
- change event and time event (case 3 in Fig. 15): time is not explicitly represented in LTS; the LTS models a system with internal activity (*e.g.* waiting for a time to elapse) without any control from its environment;
- private operation: any private operation performed under state machine control is modelled by an unobservable action, which means that for any case in Fig. 15 where operations describing effects are private, the corresponding label no longer contains the name of the operation but instead τ. This same reasoning is applied when transforming an action belonging to an activity diagram (case 6 and 7 in Fig. 15) whose operation is private.

Transformation rules can be derived from the same state as many times as there are output transitions, which suggests that UML transitions exiting a common UML state are transformed by applying rules corresponding to their types from the same LTS node. An LTS node corresponding to a UML node on which several transformations are applied is called a compositional node. Let's note that case 4 in Fig. 15 leads to two possible configurations:(1) if the transitions are of case 4 without the ELSE guard, in a combination without any other type of transition, then a deadlock node is generated; and (2) for other cases (*e.g.* type 4 without ELSE combined with case 2), the state machine cannot be considered as definitively locked since an event may trigger a transition to allow exiting the state.

The transformation may highlight inconsistent configurations, in which some transitions cannot be transformed due to the impossibility of firing. In such cases, inconsistent transitions are eliminated and a warning is issued. These kinds of configurations are described by the following cases:

- Case 1 combined with any case from 2 to 5: the *completion event* rule enforces that only the transition of case 1 can be fired; consequently, other transitions of cases 2 through 5 are eliminated.
- Cases 2 or 5 combined with case 4, including a guard and ELSE clause: again, the *completion event* rule enforces that only transitions with guards are fired; the transitions of cases 2 or 5 can never be fired and consequently get eliminated.

An activity associated with a node is represented by a sequence of actions of the *Call Operation* type: such actions are either hidden if they are internal operations, or visible if they correspond to methods that belong to a required interface. Two kinds of activities are

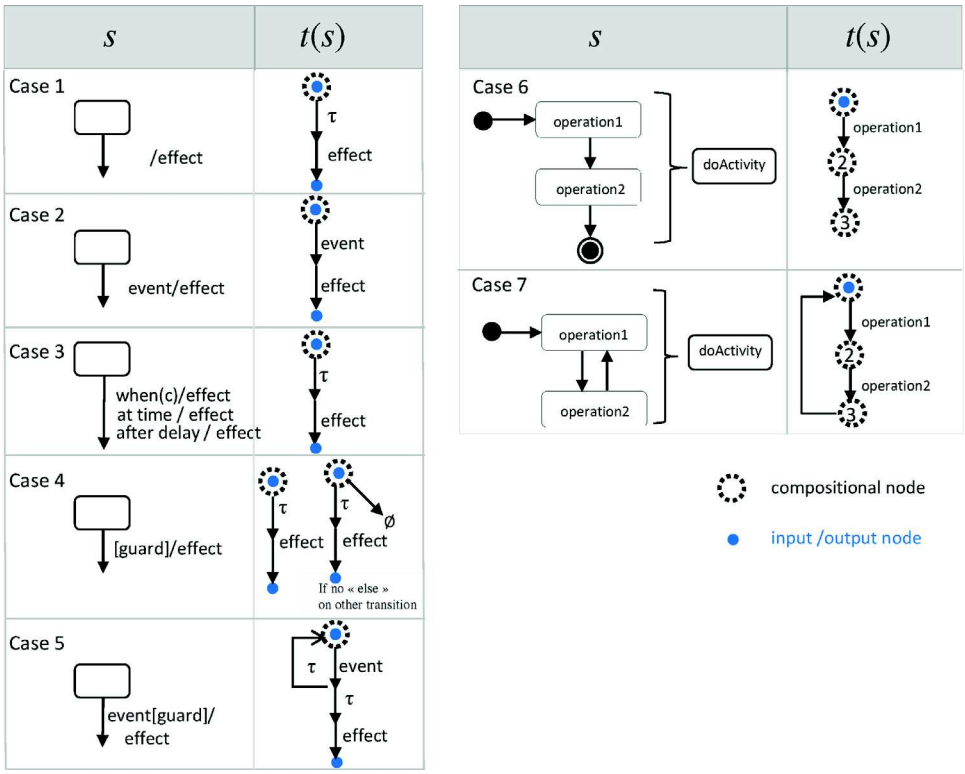


Fig. 15. UML to LTS transformation rules for (a) a state without activity (before hiding effects). (b) A state with activities.

considered in Fig. 15: activities with a final node (case 6); and activities without an end (case 7), which need to be interrupted by a trigger associated with a transition leaving the state. Case 6 features three combinational nodes hosting several transformations of output transitions: input node and node 2 hosting transformations of only the UML-triggered transitions (cases 2, 3 and 5), and node 3 hosting transformations of all UML transitions since they correspond to the end of the activity. Case 7 also contains three combinational nodes hosting just the UML-triggered transitions since a completion event can never occur.

Lastly, transforming a *submachine* node consists of distributing all output transitions onto its internal states by applying rules that depend on the type of transitions, as previously explained.

Transformation experiments have been conducted on UML state machines designed under the Topcased environment (Farail et al., 2006). Transformation of a state machine is automatically performed through a Java module implemented within our IDF framework by applying rules presented above. The LTS are generated in the Aldebaran format of CADP (Garavel et al., 2011). When several rules may be applied on a same state, a priority order is defined: cases 6 and 7 are of higher priority, cases 1 and 3 are of medium priority and cases 2, 4 and 5 are of low priority. The priority order and the mapping between states of the state machine and the LTS ensures that the transformation is deterministic. As the transformation is guaranteed to terminate, this proves that the relation $t \subseteq SM \times \mathscr{P}$ is a function $t \in SM \mapsto \mathscr{P}$. Fig. 16 provides the LTS stemming from the transformations of some state machines presented in Section 2.

7.3. Complexity and correctness of the transformation

Complexity analysis. Given a state machine with n states without concurrent region and m transitions, we deduce from the transformation rules of Fig. 15 that the time complexity of the transformation is linear and the associated LTS space is bounded by $2n + 4m$. Concurrent regions of composite states are transformed into LTS whose complexity is exponential with respect to the number of transitions whose event or actions are visible. In practice, the LTS is greatly reduced when actions are internal since a composition of τ actions produces a τ action. In other words, if the concurrent region handles only internal actions, the corresponding LTS only contains a τ transition.

Correctness of the transformation. One key point has to be discussed concerning the correctness of the abstraction from state machines to LTS. The transformation proceeds by abstraction on time and data. Moreover, the action language is limited to operation call and signal send. Other actions are considered as internal. We have to prove that our abstraction is correct, *i.e.* that any LT property p of the LTS abstraction is also a property of the state machine. The correctness property can be written:

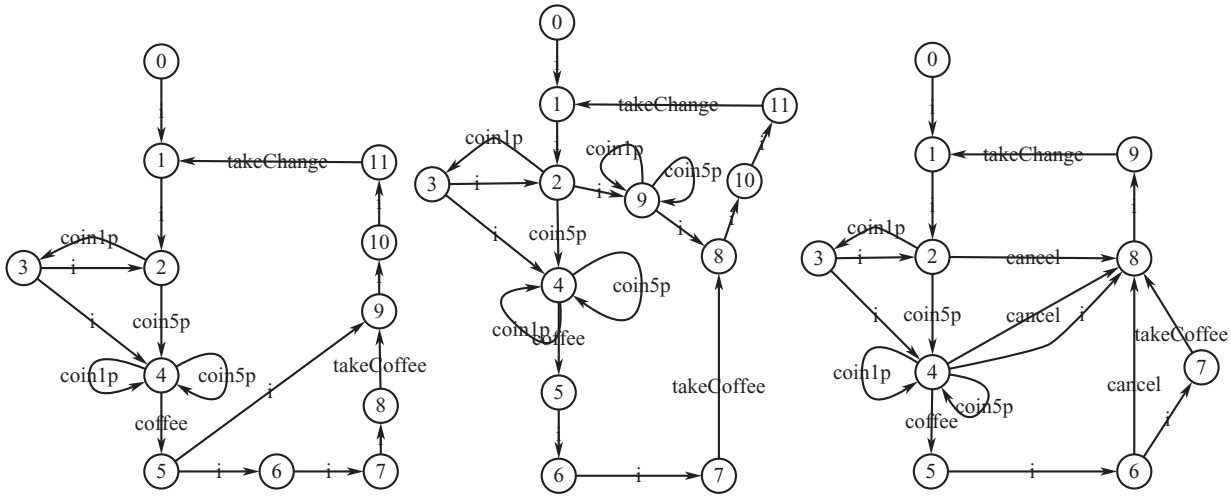
$$\text{Correctness: For any state machine } s \in SM, \quad \text{for all LT property } p, t(s) \models p \Rightarrow s \models p. \tag{1}$$

We can limit ourselves to the analysis of safety and liveness properties, since any LT property is a conjunction of both (Schneider, 1987). If, for a state machine or a LTS m, we write safety(m) and liveness(m) the sets of safety and liveness properties of m, (1) can be rewritten as follows:

$$(1) \Leftrightarrow \begin{cases} \text{safety}(s) \supseteq \text{safety}(t(s)) \\ \text{liveness}(s) \supseteq \text{liveness}(t(s)) \end{cases}$$

$$\Leftrightarrow \begin{cases} Tr(s) \subseteq Tr(t(s)) \\ \forall \sigma \in Tr(t(s)).\ Acc(s, \sigma) \subset\subset Acc(t(s), \sigma) \end{cases} \tag{2}$$

Note that, although state machines are informal, we use trace and acceptance sets notations on state machines. Trace and acceptance sets of state machines take into account data and guards, but must be written in the same language than those of LTS, which only consists of unvalued and untimed visible actions. (2) means that the transformation function t may add traces and add non-determinism (see Definitions 2 and 3 p. 20). Saying that s is more deterministic than $t(s)$, for traces of $t(s)$, also means that s has more liveness properties



a. LTS_3: CoffeeMachine b. LTS_4: AnticipatingMachine c. LTS_5: CancellableMachine

Fig. 16. LTS generated from state machines of the Coffee Machine.

than $t(s)$. Models s and $t(s)$ are finite, but $Tr(t(s))$ may be an infinite set of traces. The two conditions of (2) have to be checked on every translation pattern of Fig. 15, as well as on their compositions. Machine s and LTS $t(s)$ in cases 1 and 2 have the same traces and same acceptance sets (τ transition introduced in case 1 will have to be considered carefully when composing with other cases). In case 3, we have to make the assumption that condition c is a satisfiable formula. In some executions, c may never become true and effect would be a refused action. Nevertheless, we consider that in some circumstances and other executions, c may become true, so that effect will eventually happen. Under this assumption, s and $t(s)$ have same traces and acceptance sets. This is different for cases 4 and 5. In case 4, if guard is always true, then $Acc(s,\varepsilon) = \{\{\text{effect}\}\} \subset\subset Acc(t(s),\varepsilon) = \{\{\text{effect}\},\varnothing\}$, so that liveness(s)$\supseteq$liveness($t(s)$) and traces are the same. If guard is always false, $Acc(s,\varepsilon) = \{\varnothing\} \subset\subset Acc(t(s),\varepsilon)$. In case 5, if guard is always true, $Acc(s,\text{event}) = \{\{\text{effect}\}\} \subset\subset Acc(t(s),\text{event}) = \{\{\text{event}\},\{\text{effect}\}\}$. If guard is always false, $Acc(s,\text{event}) = \{\{\text{event}\}\} \subset\subset Acc(t(s),\text{event})$. Cases 6 and 7 are straightforward.

We now have to consider compositions of cases. Let us consider only problematic cases. Case 2 has to be considered with cases 2, 3 and 4. Only the compositions with case 4 when guard is always true or false are non trivial. If guard is always true, $Acc(s,\varepsilon) = \{\{\text{effect}\}\} \subset\subset Acc(t(s),\varepsilon) = \{\varnothing,\{\text{event}\},\{\text{effect}\}\}$. If guard is always false, $Acc(s,\varepsilon) = \{\{\text{event}\}\} \subset\subset Acc(t(s),\varepsilon)$. Similarly, case 5 composed with other cases leads to LTS having less properties. Let us consider compositions of cases 6 and 7 with 5 (other compositions are straightforward), when guard is always true or always false. If guard is always true $Acc(s,\text{event}) = \{\text{effect}\} \subset\subset Acc(t(s),\varepsilon) = \{\{\text{event}\},\{\text{operation}\},\{\text{effect}\}\}$. If guard is always false, $Acc(s,event) = \{\text{event}\} \subset\subset Acc(t(s),\varepsilon)$.

Hence, considering any cases and reasoning by induction on the composition of cases, we prove that s has more traces than $t(s)$, and that for any trace σ of s, $Acc(s,\sigma) \subset\subset Acc(t(s),\sigma)$. This establishes that s has more liveness and safety properties than $t(s)$.

The transformation t is sound, but, like many abstract interpretations, it is not complete. This means that if a property is true on the state machine, we cannot be sure it is true on the abstract LTS. For instance, in case 4 when guard is always true, s has the liveness property that it must accept {effect} whereas $t(s)$ may refuse this set. In case 5 and guard is always true, s must refuse event (safety property) whereas $t(s)$ may accept it.

Property (1) applies for LT properties of LTS. We now have to study what we can deduce from the comparison of two LTS with incremental relations.

Warnings raised by comparisons of LTS. Let us now consider two state machines s_1 and s_2 and their corresponding LTS $t(s_1)$ and $t(s_2)$. The following analysis is shown with the refinement relation, since it is the one that must preserve both safety and liveness properties. The same reasoning applies with $\sqsubseteq_{\mathsf{INC}}$ and $\sqsubseteq_{\mathsf{EXT}}$ relations, where only liveness properties could be considered. Suppose we have $t(s_1) \sqsubseteq_{\mathsf{REF}} t(s_2)$. Let's note $LT(s) = \text{safety}(s) \cup \text{liveness}(s)$ the set of safety and liveness properties of a LTS or a state machine s. For simplicity reasons, we assume here that $\sqsubseteq_{\mathsf{REF}}$ is also defined between state machines, by $s_1 \sqsubseteq_{\mathsf{REF}} s_2 =_{def} LT(s_1) \subseteq LT(s_2) \wedge \forall \sigma \in Tr(s_1) - Tr(s_2), Acc(s_1,\sigma) = \{\varnothing\}$.

From property (2) and the definition of $\sqsubseteq_{\mathsf{REF}}$, we have the following results:

$$t(s_1) \sqsubseteq_{\mathsf{REF}} t(s_2) \Rightarrow LT(t(s_1)) \subseteq LT(t(s_2)) \subseteq LT(s_2)$$
$$\text{But } t(s_1) \sqsubseteq_{\mathsf{REF}} t(s_2) \not\Rightarrow LT(s_1) \subseteq LT(s_2)$$
$$\not\Rightarrow s_1 \sqsubseteq_{\mathsf{REF}} s_2$$

We can observe that, $t(s_1) \sqsubseteq_{\mathsf{REF}} t(s_2)$ entails s_2 preserves all LT properties of $t(s_1)$, but may not preserve all LT properties of s_1. This is illustrated on Fig. 17a. State machines s_2 may not preserve unobserved properties of s_1, since in particular the verification on LTS does not analyse data nor timing aspects. When the LTS comparison does not detect any problem, the designer must remember that only untimed and unvalued traces and action sets have been analysed.

Suppose now we have $t(s_1) \not\sqsubseteq_{\mathsf{REF}} t(s_2)$, and in particular that some properties of $LT(t(s_1))$ are not in $LT(t(s_2))$. Since $LT(s_2)$ is a superset of $LT(t(s_2))$, this may be a false alarm, and we may still have $s_1 \sqsubseteq_{\mathsf{REF}} s_2$. This is illustrated on Fig. 17b. In this case, the designer can further analyse the counter-examples given by the tool (set of actions refused after some traces) on initial state machines rather than on generated LTS. He/she can then determine if the error is relevant or not.

8. Illustration of IDF relations

This part will illustrate using the example presented in Section 2 how the incremental development relations are applied. Drawing

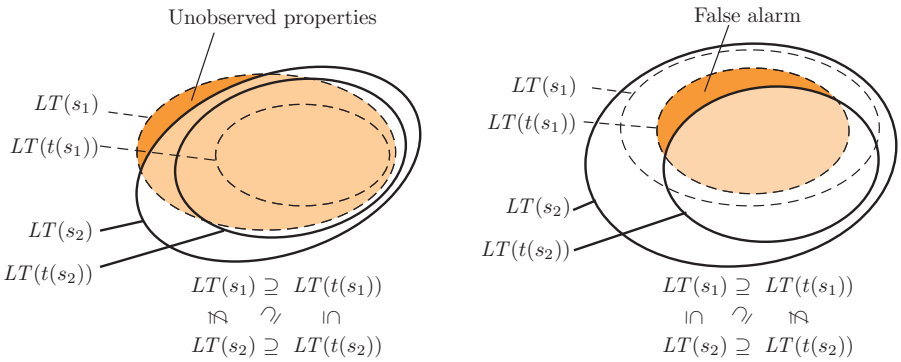
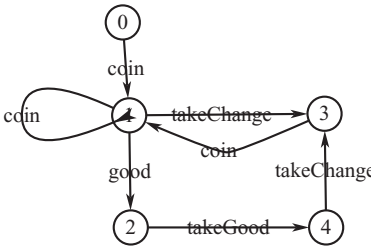


Fig. 17. Cases where (a) $t(s_1) \sqsubseteq_{\mathbf{REF}} t(s_2)$ although some properties of s_1 does not belong to s_2, (b) $t(s_1) \not\sqsubseteq_{\mathbf{REF}} t(s_2)$ is a false alarm.



State	Acceptance set
0	{{coin}}
1	{{coin}, {takeChange}, {coin, takeChange}, {coin, good, takeChange}}
2	{{takeGood}}
3	{{coin}}
4	{{takeChange}}

Fig. 18. Acceptance graph of CoffeeMachine (after label renaming).

comparisons between models requires two pre-processing operations: hiding and renaming. For this purpose, we introduce the following notations. $M_{/\{op_1,...,op_m\}}$ means that operations op_1,..., op_m are hidden in the LTS associated with the state machine M and hence are replaced by the label i. $M[op'_{1/op_1}, \ldots, op'_{n/op_n}]$ means that operations op_1,..., op_n are respectively renamed by op'_1,..., op'_n in the LTS associated with M.

8.1. Refinement verification

Hiding is necessary whenever the models to be compared have interfaces at different levels of abstraction. When the designer declares operations as 'new detailed operations', the newly detailed operations will be automatically hidden. For example, the provided interface I2 of VendingMachine will be hidden, while the inherited interface R1 remains observable. The formal verification corresponds to:

InitialMachine $\sqsubseteq_{\mathsf{REF}}$ $\mathsf{VendingMachine}_{/\{coin,good\}}$

The refinement relation is verified since after hiding, the two acceptance graphs of the state machines are identical.

Renaming is necessary whenever one operation is refined by another. When the designer uses the construction 'operations refine $<$ op $>$', refined operation op will be automatically renamed. For example, the coin operation of VendingMachine is refined by operations coin1p and coin5p in CoffeeMachine, while operation good is refined by operation coffee. The refinement relation between VendingMachine and CoffeeMachine is thus formally checked as follows:

VendingMachine $\sqsubseteq_{\mathsf{REF}}$

$\mathsf{CoffeeMachine}[coin_{/coin1p}, coin_{/coin5p}, good_{/coffee}, takeGood_{/takeCoffee}]$

Note that the operations coin and good, which were hidden during the first refinement step, no longer need to be hidden since they can be compared with respect to abstraction levels. Once again, the refinement relation is verified since after hiding, the two acceptance graphs of the state machines are identical. Fig. 18 shows the acceptance graph of these machines and its associated acceptance sets. The acceptance graph is deterministic; the non-determinism of the LTS from which it was built is translated into acceptance sets. For example, the acceptance set of State 1 in Fig. 18 indicates that the machine must accept either the coin or takeChange actions after the trace $coin^*$ and cannot refuse both of them. Furthermore, it may accept good but this is not mandatory.

The third refinement between CoffeeMachine and AnticipatingMachine does not require any hiding or renaming since their interfaces are identical. The refinement relation fails:

CoffeeMachine $\not\sqsubseteq_{\mathsf{REF}}$ AnticipatingMachine

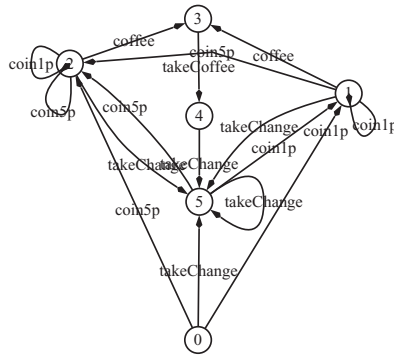
In order to analyse this failure, the first diagnosis to be performed is a conformance check since conformance is the weakest relation.

8.2. Conformance verification

The analysis of the refinement failure between CoffeeMachine and AnticipatingMachine is conducted through the verdict given by the conformance relation:

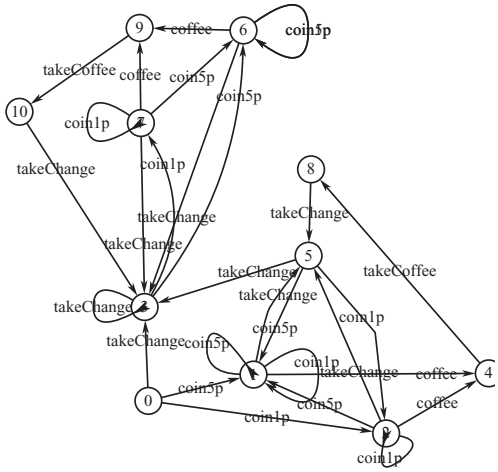
AnticipatingMachine $\not\mathsf{conf}$ CoffeeMachine

This outcome yields a set of actions that may be refused after a given trace even though they should always be accepted. In this case, failure is identified after the empty trace between the 0 nodes of the acceptance graphs associated with the AnticipatingMachine LTS and the LTS obtained from merging CoffeeMachine LTS with AnticipatingMachine LTS. The relation conf is actually implemented as a reduction between these two acceptance graphs (see Theorem 3 in Section 6). Figs. 19 and 20 show these two graphs. Comparing acceptance sets associated with the 0 nodes demonstrates that AnticipatingMachine may refuse coin1p and coin5p but cannot refuse takeChange, while CoffeeMachine cannot refuse coin1p, coin5p or



State	Acceptance set
0	{{takeChange}, {coin5p, coin1p, takeChange}}
1	{{coin5p, coin1p, coffee, takeChange}, {takeChange}, {coin5p, coin1p, takeChange}, {coin5p, coin1p, coffee}}
2	{{takeChange}, {coin5p, coin1p, takeChange}, {coin5p, coin1p, coffee}}
3	{{takeCoffee}}
4	{{takeChange}}
5	{{takeChange}, {coin5p, coin1p, takeChange}}

Fig. 19. Acceptance graph of AnticipatingMachine.



State	Acceptance set
0	{{coin5p, coin1p, takeChange}}
1	{{coin5p, coin1p, coffee, takeChange}, {takeChange}, {coin5p, coin1p, takeChange}}
2	{{coin5p, coin1p, coffee, takeChange}, {takeChange}, {coin5p, coin1p, takeChange}, {coin5p, coin1p, coffee}}
3	{{takeChange}, {coin5p, coin1p, takeChange}}
4	{{takeCoffee}}
5	{{coin5p, coin1p, takeChange}}
6	{{takeChange}, {coin5p, coin1p, takeChange}, {coin5p, coin1p, coffee}}
7	{{takeChange}, {coin5p, coin1p, coffee, takeChange}, {coin5p, coin1p, coffee}}
8	{{takeChange}}
9	{{takeCoffee}}
10	{{takeChange}}

Fig. 20. Acceptance graph obtained by merging the LTS of CoffeeMachine and AnticipatingMachine.

takeChange. This finding is formally verified by:

$$\{\{\text{takeChange}\}, \{\text{coin5p, coin1p, takeChange}\}\} \not\subset \{\{\text{coin5p, coin1p, takeChange}\}\}$$

8.3. Extension verification

Lastly, we must verify the extension relation between CancellableMachine and CoffeeMachine. An extension verification does not require any hiding of operations even if the interfaces are different. It is simply necessary to check that new operations imply new traces without corrupting the expected traces. The relation is formally verified as follows:

$$\text{CoffeeMachine} \sqsubseteq_{\text{EXT}} \text{CancellableMachine}$$

In this case, the relation is verified through an examination of acceptance graphs and set inclusion.

8.4. Overview of incremental development of the coffee machine

Fig. 21 offers a synthesis of the relations verified between the state machines of Section 2. Let's note that due to property P_5 regarding the composability of incremental relations, the fact that $\text{InitialMachine} \sqsubseteq_{\text{INC}} \text{CancellableMachine}$ can be deduced.

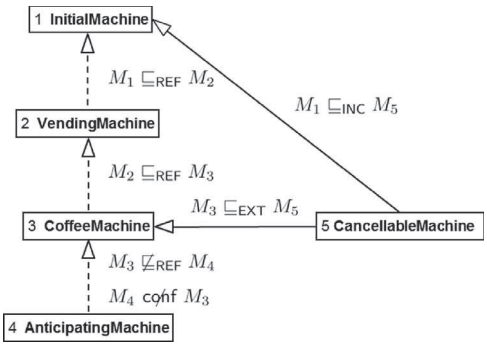


Fig. 21. Flowchart of the incremental development verification process of the coffee machine.

CancellableMachine covers the first-level requirements. At this modelling stage, the secondary requirements such as maintenance need to be taken into consideration. Should these requirements be

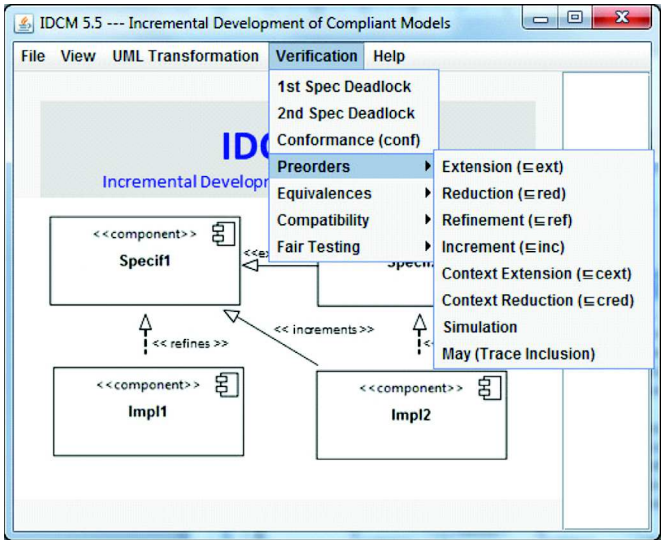


Fig. 22. IDCM User interface for model verification.

evaluated as optional, model development can stop and a product line can be developed and distributed. The benefit of this approach is to postpone the development of new functionalities that will lead to new system releases, by taking advantage of previously developed models. When secondary requirements are mandatory, the model extension must continue to cover all requirements.

9. IDCM: a tool supporting IDF

This part will provide an overview of the tool we developed to support IDF. This development work was performed in Java. The tool comprises two main modules: the transformation of UML models into LTS, and model analysis. The transformation module consists of parsing primitive components constituting a UML model and then transforming them into LTS according to the rules presented in Section 7. This transformation involves the Java DOM (Document Object Model) technology parser. The LTS are generated in the textual CADP Aldebaran format (Garavel et al., 2011). It is thus possible to apply CADP tools to reduce large LTS (the bcg_min tool) and produce the LTS of the component assembly, having the LTS associated with each component.

The verification step consists of comparing two models according to a relation selected by the designer, as depicted in Fig. 22. Once the relation is verified, a diagnosis is provided to the designer. In case of failure, an explanation is given in terms of rejected actions. In Fig. 23 for example, IDCM indicates that AnticipatingMachine is not a refinement of CoffeeMachine since it may initially accept the takeChange action while refusing coin1p and coin5p.

A number of experiments have been conducted by the team members and students to validate IDCM; moreover, case studies have been established from conventional examples stemming from the state-of-the-art such as the ATM (Automatic Teller Machine), JobShop (Milner, 1989), Phone System (Luong et al., 2008), AFL (Adaptive Forward Lighting System) and VendingMachine (Courbis et al., 2012). The development processes of these studies contain approximatively 5 to 10 steps. Experiments pointed out that extending or refining state machines is not intuitive even on simple models with few states, even for team members that master the concepts of incremental development. Our technique discovered true errors in designed state machines, that had not been intentionally added. IDCM gave us verdict traces, which were useful to identify design errors in state machines.

The IDCM performance on the JobShop models built from the example proposed by Milner (1989) will be presented below. The model has been set up using an incremental approach, starting from a simple specification that has been extended and refined into eight steps. In order to evaluate the time efficiency of IDCM for computing the relations, large LTS are requisite. As we have seen, LTS size is linear with respect to state machine size. It is thus difficult to obtain large LTS from actual state machines. Consequently, we have designed state machines for four simple components whose specification is given below and built up large models by combining their LTS in different configurations. The LTS generated from the composition of the combined LTS has been obtained through the CADP toolbox by using

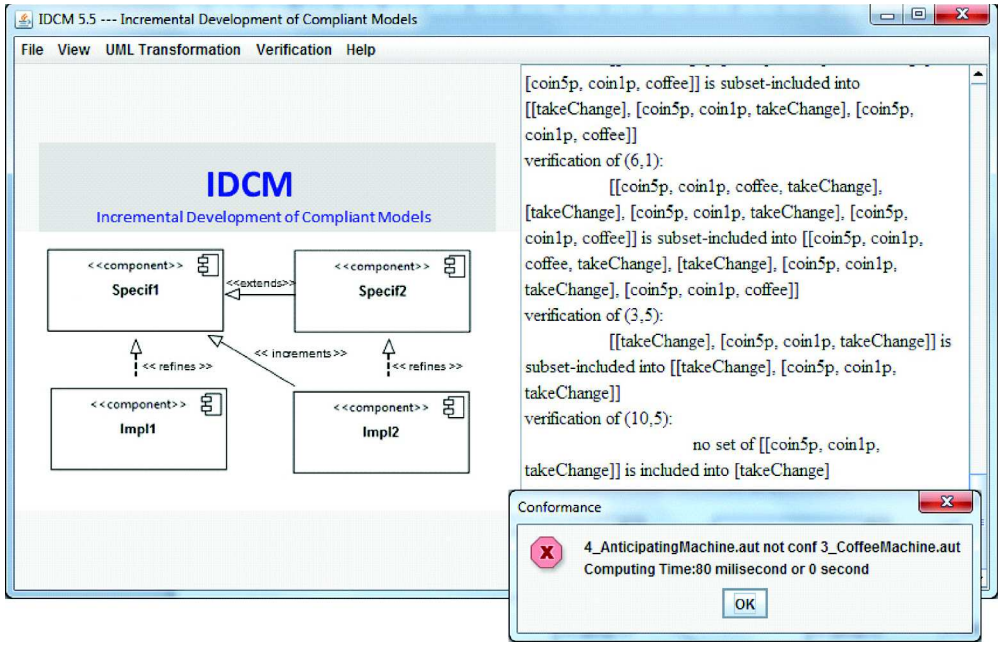


Fig. 23. IDCM diagnosis.

Table 7
Performance of IDCM on JobShop models.

System model	Specification	Components	Transitions	States	Relation	Verification time (s)
Workstation1 (M_1)	One jobber working with two tools	Jobber, Tool	88	45	$S_1 \sqsubseteq_{\mathbf{REF}} M_1$	12
Workshop1 (M_2)	Two Workstation1 in parallel	Workstation1, Router	13,224	3,249	$M_1 \sqsubseteq_{\mathbf{EXT}} M_2$ $S_2 \sqsubseteq_{\mathbf{REF}} M_2$	13 10
Workstation2 (M_3)	Two jobbers sharing two tools	Jobber, Tool, Router	2,511	815	$M_2 =_{\mathbf{ct}} M_3$	13
Line1 (M_4)	Sequence of two Workstation1	Workstation1, Glue	23,387	5,673	$M_3 \sqsubseteq_{\mathbf{EXT}} M_4$	9
Workshop2 (M_5)	Two Workstation2 in parallel	Workstation2	4,092,930	664,225	$M_4 \sqsubseteq_{\mathbf{EXT}} M_5$	24
Line2 (M_6)	Sequence of Workstation1 and Workstation2	Workstation1, Workstation2, Glue	510,526	100,865	$M_5 =_{\mathbf{ct}} M_6$	12
Workshop3 (M_7)	Four Workstation1 in parallel	Workstation1	32,076,000	4,100,625	$M_6 =_{\mathbf{ct}} M_7$	310

the operator of synchronous interaction of processes (Garavel et al., 2011).

Jobber:	takes a job, asks for a tool if necessary and puts the tool away when the job is completed.
Tool:	offers two actions: get when available and put when busy. This component can only be used by one jobber at a time.
Router:	dispatches inputs to sub-components and collects outputs from these sub-components.
Glue:	connects an output job with an input job.

The first specification, called Spec1Job, defines a system that initially waits for a single job, produces an output job and returns to its initial state. Its extension, called Spec2Jobs, defines a system that accepts one or two input jobs (potentially arriving in parallel) and consequently produces one or two output jobs. The refinement of these specifications involves two main primitive components, Jobber and Tool, and two auxiliary primitive components, Glue and Router. The specification of these components is presented in Table 7.

Fig. 24 shows the models that have been built and their development relationships. Details on components and the size of the generated LTS are listed in Table 7 along with the time consumed for their relation verification. This time value includes a minimisation using CADP and the relation computation with IDCM. All experiments have been conducted on a Windows 7 PC with 64 bits and 2.70 GHz.

These experiments indicate that IDCM verifications can be performed on reasonably large-sized LTS. Note that, for Line1 and Line2, verification times remain low. This is due to the fact that the corresponding minimised LTS are of same size as Workshop1. Concerning Workshop2 and Workshop3, verification times mainly consist of generation and minimisation times and corresponding LTS are drastically reduced. The systems studied herein have been built as an assembly of active components. When incrementally designing such assemblies, it might be assumed that only modified components are to be verified: this consideration lies outside the scope of this paper and will be studied in a subsequent work.

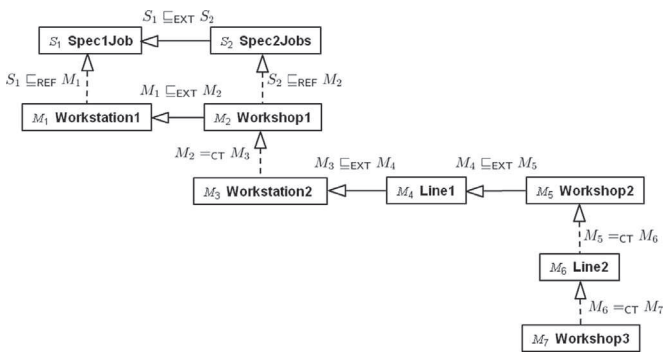


Fig. 24. Relations verified between the models presented in Table 7.

10. Concluding discussion and future work

We have defined IDF, a framework for incremental development of UML state machine models. This approach combines the advantages of vertical and horizontal developments. Vertically, it supports refinement approaches from abstractions to realisations, while horizontally it serves to enhance models and integrate new behaviours so as to easily and safely develop intermediate product versions.

Main benefits of IDF are the following:

- The construction is separated into small tractable steps.
- IDF offers evaluation means *during* the construction of models, and not only at the end the design phase. This "pocket-verification" technique is therefore an ongoing evaluation means for designers, but it is also a solution to verification problems which are intractable when models become too large. In particular, the verifications we propose for reactivity aspects concern the preservation of 'must-behaviours', which belong to the class of liveness properties. Such liveness properties are known to be impossible to analyse by testing techniques, and many model checking techniques ignore them.
- Designers can verify models without having to use other modelling languages to describe required properties. There are two drawbacks to a separate description of required properties of the system with a specific language: (i) it requires modelling skills with a formal language suited to describe requirements (ii) it needs validation means to ensure that formal requirements actually correspond to the informal requirements.
- IDF offers a formal support to an agile approach, allowing engineers to develop first simplified or prototype versions and to enrich them afterwards. Such versions can be reviewed by stakeholders before the system is achieved.
- IDF supports the development of a line of products. Intermediate models can be extended and refined in various ways to develop several kinds of products.

IDF is based on a set of formal relations for refinement, extension and increment defined on a formal abstract model of labelled transition systems. By this way, the verification we propose belongs to model-checking techniques, since it is based on exhaustive state space exploration of a finite state system (Bhaduri, 2004). Usually, model-checking techniques compare behavioural models with required properties, described for instance in temporal logics. Our model-comparison approach has several limitations compared to such techniques:

- First model is not verified and similarly, new added behaviours are not verified with respect to their requirements. We only verify that they do not contradict existing behaviours.
- UML state machine language is not as expressive as temporal logics to describe required properties.
- The safety verification we propose, in case of vertical refinement developments, consider that all non described behaviours in the initial model are unwanted behaviours. An advantage of separately describing some safety properties is to enable the designer to extend the models and to verify that precisely described safety properties are still satisfied.

In order to reduce these drawbacks, our approach could be completed by techniques allowing the designer to describe unexpected behaviours, such as a kind of 'anti state machine' describing an unwanted set of behaviours. We can also consider other UML diagrams, such as sequence diagrams, which could be used to describe negative scenarios corresponding to unexpected behaviours. Negative scenarios are used for instance in Ramchandani (2009). At last, we have shown that the LTS generated as an abstraction of state machines have all their safety and liveness properties included inside originating state machine's properties. Designers can therefore take benefit of other model-checking facilities proposed by a tool such as CADP.

At the present time, we are working on the incremental design of architectures defined by assemblies of active components. In a first approach, this work could require to compute the overall behaviour of architectures from the behaviour of their components and their interconnections. It is necessary to formally define the transformation of architectures into LTS. In a second approach, we could consider the substitution question. Indeed, an interesting question concerning architectures is the one of comparing two architectures where only one of their components differs.

Future steps of our work will also be focused on building a UML profile that fits the IDF paradigms. In this manner, the designer may express expected model developments with respect to refinement and extension in implying the automatic verification of relations. Furthermore, IDF may be enhanced by a methodology to guide designers according to requirement priority levels or development strategies.

References

Abrial, J.-R., 1996. The B-Book — Assigning Programs to Meanings. Cambridge University Press.

Abrial, J.-R., 2010. Modeling in Event-B — System and Software Engineering. Cambridge University Press.

Aldini, A., Bernardo, M., Corradini, F., 2010. A Process Algebraic Approach to Software Architecture Design. Springer doi:10.1007/978-1-84800-223-4.

Alpern, B., Schneider, F., 1985. Defining liveness. Inf. Process. lett. 21 (October), 181–185.

Alpern, B., Schneider, F.B., 1987. Recognizing safety and liveness. Distrib. Comput. 2 (3), 117–126. doi:10.1007/BF01782772.

Ambler, S.W., 2008. The Object Primer, Agile Model-Driven Development with UML 2.0. Cambridge edition.

Baader, F., Calvanese, D., McGuinness, D.L., 2003. The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press.

Baier, C., Katoen, J., 2008. Principles of model checking. MIT Press.

Bertot, Y., Castéran, P., 2004. Interactive Theorem Proving and Program Development. Springer Science & Business Media.

Bhaduri, P., 2004. Model checking of statechart models. arXiv cs/SE 0407038, 1–41.0407038v1

Bloom, B., Istrail, S., Meyer, A.R., 1995. Bisimulation can't be traced. J. ACM (JACM) 42 (1), 232–268.

Boiten, E.A., Bujorianu, M.C., 2003. Exploring UML refinement through unification. In: Jürgens, J., Rumpe, B., France, R., Fernandez, E. (Eds.), Critical Systems Development with UML—Proceedings of the UML'03 workshop. Technische Universität München, pp. 47–62.

Bolognesi, T., Brinksma, E., 1987. Introduction to the ISO specification language LOTOS. In: Computer Networks and ISDN systems, Vol. 14, pp. 25–59.

Bolton, C., Davies, J., 2002. A comparison of refinement orderings and their associated simulation rules. Electron. Theor. Comput. Sci. 70 (3), 297–310.

Bozga, M., Graf, S., Mounier, L., 2002. IF-2.0: a validation environment for component-based real-time systems. In: Brinksma, E., Larsen, K. (Eds.), Conference on Computer Aided Verification (CAV). Springer, Berlin, Heidelberg, pp. 343–348.

Brinksma, E., Scollo, G., 1986. Formal Notions of Implementation and Conformance in LOTOS. Technical Report. Twente University of Technology, Department of Computer Science. Enschede.

Brookes, S., 1984. A theory of communicating sequential processes. J. Assoc. Comput. Mach. 31 (3), 560–599.

Burmester, S., Giese, H., Hirsch, M., Schilling, D., 2004. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (Eds.), Workshop on Specification and Validation of UML Models for Real-Time and embedded Systems (SVERTS). Springer Verlag, pp. 1–20.

Chimisliu, V., Schwarzl, C., Peischl, B., 2009. From UML statecharts to LOTOS : a semantics preserving model transformation. In: Choi, B. (Ed.), 9th Conference on Quality Software (QSIC'2009), pp. 173–178. doi:10.1109/QSIC.2009.31.

Cleaveland, R., Hennessy, M., 1993. Testing equivalence as a bisimulation equivalence. Form. Asp. Comput. 5, 1–20.

Cleaveland, R., Steffen, B., 1990. A preorder for partial process specifications. In: Baeten, J., Klop, J. (Eds.), CONCUR '90 Theories of Concurrency: Unification and Extension. Springer-Verlag, New York, Inc., pp. 141–151.

Corporation, O., 2013. The java tutorials — trial essential classes: concurrency. Liveness.

Courbis, A.-L., Lambolais, T., Luong, H.-V., Phan, T.-L., Urtado, C., Vauttier, S., 2012. A formal support for incremental behavior specification in agile development. In: The 24th International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute Graduate School, USA, pp. 694–699.

Crane, M.L., Dingel, J., 2005. On the Semantics of UML State Machines: Categorization and Comparison. Technical Report. School of Computing, Queen's University, Kingston, Ontarion Canada.

Dragomir, I., 2014. Contract-based modeling and verification of timed safety requirements for system design in SysML. Toulouse 3.Ph.D. Thesis

Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., Pantel, M., 2006. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. Ingénieurs de l'Automobile 781, 54–59.

Fecher, H., Schönborn, J., 2007. UML 2.0 state machines: complete formal semantics via core state machine. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (Eds.), Formal Methods: Applications and Technology. Springer, Berlin Heidelberg, pp. 244–260. doi:10.1007/978-3-540-70952-7_16.

Fecher, H., Schönborn, J., Kyas, M., de Roever, W., 2005. 29 new unclarities in the semantics of UML 2.0 state machines. In: Lau, K.-K., Banach, R. (Eds.), Formal Methods and Software Engineering. Springer, Berlin, Heidelberg, pp. 52–65.

Fernandez, J.-C., Mounier, L., 1992. On the fly verification of behavioural equivalence and preorders. In: Computer Aided Verification. Springer, Berlin, Heidelberg, pp. 181–191.

Garavel, H., Lang, F., Mateescu, R., Serwe, W., 2011. CADP 2010: a toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. In: Lecture Notes in Computer Science, Vol. 6605. Springer, Berlin, Heidelberg, pp. 372–387. doi:10.1007/978-3-642-19835-9_33.

Gnesi, S., Mazzanti, F., 2005. A model checking verification environment for UML statecharts. In: XLIII Congresso Annuale AICA, p. 10.

Goldsmith, M., Zakiuddin, I., 1999. Critical systems validation and verification with CSP and FDR. In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (Eds.), Applied Formal Methods – FM-Trends 98. Springer, Berlin, Heidelberg, pp. 243–250. doi:10.1007/3-540-48257-1_15.

Halbwachs, N., 1992. Synchronous programming of reactive systems. Springer Science & Business Media.

Hennessy, M., 1988. Algebraic theory of processes. MIT Press.

Hoare, C.A.R., 2004. Communicating Sequential Processes. Prentice Hall International.

Holzmann, G.J., 1997. The model checker SPIN. IEEE Trans. Softw. Eng. 23 (5), 279–295.

Hudon, S., Hoang, T.S., 2013. Systems design guided by progress concerns. In: Johnsen, E., Petre, L. (Eds.), Integrated Formal Methods. Springer, Berlin, Heidelberg, pp. 16–30.

Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L., 2005. Consistency problems in UML-based software development. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (Eds.), UML Modeling Languages and Applications. Springer, Berlin, Heidelberg, pp. 1–12. doi:10.1007/978-3-540-31797-5_1.

ISO/IEC9646, 1991. Information technology – open systems interconnection – conformance testing methodology and framework – part 1: general concepts.

Jard, C., Jéron, T., 2005. TGV: theory, principles and algorithms. In: International Journal on Software Tools for Technology Transfer (STTT), Vol. 7. Springer Verlag, pp. 297–315. doi:10.1007/s10009-004-0153-x.

Katoen, J., 2012. Model checking: one can do much more than you think!. In: Fundamentals of Software Engineering. Springer, Berlin, Heidelberg, pp. 1–14.

Khalil, A., Dingel, J., 2013. Supporting the Evolution of UML Models in Model Driven Software Development : A Survey. Technical Report 602. School of Computing, Queen's University, Ontario, Canada.

Khendek, F., von Bochmann, G., 1995. Merging behavior specifications. Form. Methods in Syst. Des. 6 (3), 259–293.

Knapp, A., Merz, S., Rauh, C., 2002. Model checking timed UML state machines and collaborations. In: Damm, W., Olderog, E.-R. (Eds.), 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002). Springer, Berlin, Heidelberg, pp. 395–414. doi:10.1007/3-540-45739-9_23.

Knapp, A., Mossakowski, T., 2014. An institution for simple UML state machines. arXiv preprint cs 1411.4495, 1–24.

Kouchnarenko, O., Lanoix, A., 2006. How to Verify and Exploit a Refinement of Component-based Systems. Technical Report. INRIA - n5898.

Kupferman, O., Vardi, M., 2001. Model checking of safety properties. Form. Methods in Syst. Des. 19 (3), 291–314.

Lano, K. (Ed.), 2009. UML 2 Semantics and Applications. John Wiley and Sons, Inc., Hoboken, New Jersey.
Larman, G., Basili, V.R., 2003. Iterative and incremental development: a brief history. Computer 36 (6), 47–56.
Latella, D., Majzik, I., Massink, M., 1999. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Form. Asp. Comput. 11 (6), 637–664. doi:10.1007/s001659970003.
Leduc, G., 1991. On the Role of Implementation Relations in the Design of Distributed Systems Using LOTOS. University Liège Ph.D. thesis.
Leduc, G., 1992. A framework based on implementation relations for implementing LOTOS specifications. In: Computer Networks and ISDN Systems, Vol. 25, pp. 23–41.
Leduc, G., 1995. Failure-based congruences, unfair divergences and new testing theory. In: Vuong, S., Chanson, S. (Eds.), Proceedings of the Fourteenth of a Series of Annual Meetings on Protocol Specification, Testing and Verification. Springer, US, pp. 252–267. doi:10.1007/978-0-387-34867-4_17.
Lilius, J., Paltor, I.P., 1999. Formalising UML state machines for model checking. In: France, R., Rumpe, B. (Eds.), UML'99 — The Unified Modeling Language. Springer, Berlin, Heidelberg, pp. 430–444. doi:10.1007/3-540-46852-8_31.
Liu, S., Liu, Y., André, E., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S., 2013. A Formal Semantics for the Complete Syntax of UML State Machines with Communications. Technical Report. School of Computing, National University of Singapore Republic.
LiuS. and Liu, Y. and André, E. and Choppy, C. and Sun, J. and Wadhwa, B. and Dong, J., 2013. A formal semantics for complete uml state machines with communications. In: Johnsen, E., Petre, L. (Eds.), Integrated Formal Methods. In: Lecture Notes in Computer Science, Vol. 7940. Springer, Berlin, Heidelberg, pp. 331–346. doi:10.1007/978-3-642-38613-8_23.
Lowe, G., 2008. Specification of communicating processes: temporal logic versus refusals-based refinement. Form. Asp. Comput. 20 (3), 277–294. doi:10.1007/s00165-007-0065-0.
Lucas, F.J., Molina, F., Toval, A., 2009. A systematic review of UML model consistency management. Inf. Softw. Technol. 51 (12), 1631–1645. doi:10.1016/j.infsof.2009.04.009.
Luong, H.-V., 2010. Construction incrémentale de spécifications de systèmes critiques intégrant des procédures de vérification. Université Paul Sabatier Toulouse III Ph.D. thesis.
Luong, H.-V., Lambolais, T., Courbis, A.-L., 2008. Implementation of the conformance relation for incremental development of behavioural models. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (Eds.), Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). Springer, Berlin, Heidelberg, pp. 356–370. doi:10.1007/978-3-540-87875-9_26.
Manual FDR2, 2010. Failures-Divergence Refinement. Oxford University Computing Laboratory.
Meng, S., Naixiao, Z., Barbosa, S., 2004. On semantics and refinement of UML statecharts: a coalgebraic view. In: Software Engineering and Formal Methods (FSEN). IEEE, pp. 164–173.
Mens, T., Van Der Straeten, R., Simmonds, J., 2005. A framework for managing consistency of evolving UML models. In: Yang, H. (Ed.), Software Evolution with UML and XML. Idea Group Inc. Publishing, p. 30. doi:10.4018/978-1-59140-462-0.ch001.
Milner, R., 1989. Communication and concurrency. Prentice-Hall, Inc..
Milner, R., 1999. Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press.
Moseley, S., Randall, S., Wiles, A., 2006. In pursuit of interoperability. In: Jakobs, K. (Ed.), Advanced Topics in Information Technology Standards and Standardization Research. Idea Group Publishing, Hershey, pp. 321–323. doi:10.4018/978-1-59140-938-0.ch017.
Ober, I., Graf, S., Ober, I., 2006. Validating timed UML models by simulation and verification. In: International Journal on Software Tools for Technology Transfer (STTT), Vol. 8, pp. 128–145. doi:10.1007/s10009-005-0205-x.
Pons, C., 2005. On the definition of UML refinement patterns. In: Proceedings of tne 2nd Model Design and Validation (MoDeVa) Workshop, Montego Bay, Jamaica, p. 6.
Prochnow, S., Schaefer, G., Bell, K., Von Hanxleden, R., 2006. Analyzing robustness of UML state machines. In: Gerard, S., Graf, S., Haugen, O., Ober, I., Selic, B. (Eds.), International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems, pp. 61–80.
Puhakka, A., Valmari, A., 2001. Liveness and fairness in process-algebraic verification. In: Larsen, K.G., Nielsen, M. (Eds.), CONCUR 2001 — Proceedings of the 12th International Conference on Concurrency Theory, Aalborg, Denmark, August 20-25. Springer, Berlin, Heidelberg, pp. 202–217. doi:10.1007/3-540-44685-0_14.
Ramchandani, D., 2009. Refining Labelled transition systems using scenario-based specifications. Technical Report. Imperial College London, Department of Computing.
Rasch, H., Wehrheim, H., 2003. Checking consistency in UML diagrams: Classes and state machines. In: Formal Methods for Open Object-Based Distributed Systems, pp. 229–243.
Royer, J.-C., 2003. Temporal logic verifications for UML: the vending machine example. Proceedings of 4th Rigourous Object-Oriented Methods Workshop; RSTI L'objet 9 (04), 73–92.
Ruhroth, T., Wehrheim, H., 2012. Model evolution and refinement. Sci. Comput. Program. 77 (3), 270–289. doi:10.1016/j.scico.2011.04.007.
Sagonas, K. (Ed.), 2013. Practical Aspects of Declarative Languages. number 7752 in Lecture Notes in Computer Science, Springer Verlag.
Said, M., Butler, M., Snook, C., 2009. Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti, A., Dams, D. (Eds.), FM 2009: Formal Methods SE - 37. In: Lecture Notes in Computer Science, Vol. 5850. Springer, Berlin, Heidelberg, pp. 579–595. doi:10.1007/978-3-642-05089-3_37.
Said, M.Y., 2010. Methodology of Refinement and Decomposition in UML-B. University of Southampton Ph.D. thesis.
Said, M.Y., Butler, M., Snook, C., 2013. A method of refinement in UML-B. Softw. Syst. Model. 1–24. doi:10.1007/s10270-013-0391-z.
Schneider, F.B., 1987. Decomposing Properties into Safety and Liveness Using Predicate Logic. Technical Report. Cornell University Ithaca, NY, Dept. of Computer Science.
Schneider, S., Treharne, H., Wehrheim, H., 2014. The behavioural semantics of Event-B refinement. Form. Asp. Comput. 26 (2), 251–280.. doi:10.1007/s00165-012-0265-0.
Scholz, P., 2001. Incremental design of statechart specifications. Sci. Comput. Program. 40, 119–145.
Schönborn, J., Kyas, M., 2010. Refinement patterns for Hierarchical UML state machines. In: Arbab, F., Sirjani, M. (Eds.), Third International conference on Fundamentals of Software Engineering. In: Lecture Notes in Computer Science, Vol. 5961. Springer, Berlin, Heidelberg, pp. 371–386. doi:10.1007/978-3-642-11623-0_22.
Smith, G., 2000. The object-Z specification language, Vol. 1. Advances in Formal Methods. Kluwer Academic Publishers, Boston, MA doi:10.1007/978-1-4615-5265-9.
Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M., 2001. Refactoring UML models. In: Gogolla, M., Kobryn, C. (Eds.), UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Springer, Berlin, Heidelberg, pp. 134–148. doi:10.1007/3-540-45441-1_11.
Tretmans, J., 1999. Testing concurrent systems: a formal approach. In: Baeten, J., Mauw, S. (Eds.), CONCUR'99 Theories of Concurrency: Unification and Extension. Springer, Berlin, Heidelberg, pp. 46–65. doi:10.1007/3-540-48320-9_6.
Truong, N.-T., Souquieres, J., 2005. Verification of behavioural elements of UML models using B. In: Proceedings of the 2005 ACM symposium on Applied Computing. ACM, New York, NY, USA, pp. 1546–1552. doi:10.1145/1066677.1067024.
Usman, M., Nadeem, A., Kim, T.H., Cho, E.S., 2008. A survey of consistency checking techniques for UML models. In: Proceedings of the 2008 Advanced Software Engineering and its Applications, pp. 57–62. doi:10.1109/ASEA.2008.40.
Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V., 2003. Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (Eds.), UML 2003 — The Unified Modeling Language. Modeling Languages and Applications. Springer, Berlin, Heidelberg, pp. 326–340. doi:10.1007/978-3-540-45221-8_28.
Van Glabbeek, R., Ploeger, B., 2008. Five determinisation algorithms. In: Ibarra, O.H., Ravikumar, B. (Eds.), International Conference Implementation and Application of Automata. Springer, Berlin, Heidelberg, pp. 161–170. doi:10.1007/978-3-540-70844-5_17.

Thomas Lambolais had a PhD in Computer Science by Institut National Polytechnique de Lorraine (France) in 1997. From 1998 to 2001, he worked at TRT (Thalès Research and Technology). Since 2001, he is associate professor at the Ecole des mines d'Alès (France). His research interests are in software engineering and include requirements engineering. He is interested in methods and tools to support incremental development of behavioural models based on verification techniques.

Anne-LiseCourbis had a PhD in Computer Science by University of Montpellier 2 (France), in 1991. From 1992 to 1996, she worked as associate professor at LERI - EERIE (Nîmes, France). Since 1996, she is associate professor at the Ecole des mines d'Alès (France). Her research interests are in system engineering and include discrete behavioural modelling and simulation. She contributes to the incremental development of behavioural models based on verification techniques.

Hong-Viet Luong had a PhD in Computer Science by University Paul Sabatier of Toulouse 3 (France), in 2010. His area of interest is incremental development of UML state machines. From 2011 to 2013, he had a post-doctoral position at the AMPERE laboratory (INSA Lyon, France. Since 2014, he is R&D Engineer at the M2M-NDT company (France).

Christian Percebois is professor of computer science at the University of Toulouse since 1992. He was always interested in software engineering. He worked on Lisp and Prolog interpreters, garbage collecting for symbolic computations, asynchronous backtrackable communications in parallel logic languages, abstract machine construction through operational semantics refinements, typing in object-oriented programming and multiset rewriting techniques in order to coordinate concurrent objects. Today his main research tries to combine formal methods and software engineering, in particular for graph rewriting systems.